kProbLog: an algebraic Prolog for machine learning

Noname manuscript No.
(will be inserted by the editor)

kProbLog: an algebraic Prolog for machine learning

Francesco Orsini · Paolo Frasconi · Luc De Raedt

Received: date / Accepted: date

Abstract We introduce kProbLog as a declarative logical language for machine learning. kProbLog is a simple algebraic extension of Prolog with facts and rules annotated by semiring labels. It allows to elegantly combine algebraic expressions with logic programs. We introduce the semantics of kProbLog, its inference algorithm, its implementation and provide convergence guarantees.

We provide several code examples to illustrate its potential for a wide range of machine learning techniques. In particular, we show the encodings of state-of-the-art graph kernels such as Weisfeiler-Lehman graph kernels, propagation kernels and an instance of Graph Invariant Kernels (GIKs), a recent framework for graph kernels with continuous attributes. However, kProbLog is not limited to kernel methods and it can concisely express declarative formulations of tensor-based algorithms such as matrix factorization and energy-based models, and it can exploit semirings of dual numbers to perform algorithmic differentiation. Furthermore, experiments show that kProbLog is not only of theoretical interest, but can also be applied to real-world datasets.

At the technical level, kProbLog extends aProbLog (an algebraic Prolog) by allowing multiple semirings to coexist in a single program and by introducing meta-functions for manipulating algebraic values.

Keywords algebraic Prolog, kernel programming, graph kernels, machine learning

Francesco Orsini, E-mail: francesco.orsini@cs.kuleuven.be
Department of Computer Science, Katholieke Universiteit Leuven
Department of Information Engineering, Università degli Studi di Firenze

Paolo Frasconi, E-mail: paolo.frasconi@unifi.it
Department of Information Engineering, Università degli Studi di Firenze

Luc De Raedt, E-mail: luc.deraedt@cs.kuleuven.be
Department of Computer Science, Katholieke Universiteit Leuven

1 Introduction

The field of logical and relational learning has already a long tradition, cf. [50, 11, 43]. In the '80s and '90s, the goal of this field was to use purely logical and relational representations within machine learning and in this way, provide more expressive representations that allow complex datasets and background knowledge to be represented. The key challenge at the time was to tightly integrate these representations with symbolic machine learning methods that were then popular, such as rule-learning and decision trees [54]. But the field of machine learning has evolved and broadened; in the last two decades it has focused more on statistical and probabilistic approaches, on kernel and support vector machines and on neural networks. These trends in machine learning have inspired logical and relational learning researchers to extend their goals and to investigate how logical and relational learning principles can be exploited within probabilistic methods, kernel methods, and neural networks.

This is best illustrated by the success of statistical relational learning and probabilistic programming [13, 27], which combine logical and relational learning and programming with probabilistic graphical models. Today there exist many frameworks and formalisms that tightly integrate these two paradigms; they support probabilistic and logical inference as well as learning. Prominent examples include PRISM [52], Dyna [19, 18], Markov Logic [49], BLOG [42], and ProbLog [12]. Statistical relational learning and probabilistic programming have enabled an entirely new generation of applications.

While there has been a lot of research on integrating probabilistic and logic reasoning, the combination of kernel-based methods with logic has been much less investigated with the notable exceptions of kLog [22], kFOIL [39] and Gärtner et al's work [25, 26]. kLog is a relational language for specifying kernel-based learning problems. It produces a graph representation of a relational learning problem in the spirit of knowledge-based model construction and then employs a graph kernel on the resulting representation. kFOIL is a variation on the rule learner FOIL [48] that can learn kernels defined as the number of clauses that succeed in both interpretations. Gärtner et al developed kernels within a typed higher order language and used it on some inductive logic programming benchmarks.

Also for what concerns neural networks, there is a stream of research work that combines neural with logical and symbolic representations, which is often referred to as neural-symbolic learning and reasoning [23, 24].

This research on probabilistic models, kernel-based methods and neural networks shows that it is important for logical and relational learning to integrate its principles and techniques with those of other schools in machine learning. Furthermore, the power of logical and relational learning is not only concerned with the expressiveness of the logical and relational representations but also with their declarativeness. Indeed, it has been repeatedly argued that logical and relational learning allows one to declaratively specify and solve problems by specifying background knowledge and declarative bias [11, 43]. This property of logical and relational learning has turned out to be essential for many successes in applications as making small changes to the background knowledge or bias allows one to easily control the learning algorithm. While in the above mentioned probabilistic, kernel-based and neural approaches to logical and relational learning, it is typically possible to tune the logical and relational part in a declarative way, the probabilistic,

kernel or neural components are typically built-in and hardcoded into the underlying formalisms and are very hard to modify. For instance, kLog was designed to allow different graph kernels to be plugged in, but support to declaratively specify the kernel is missing. Standard probabilistic programming languages such as PRISM and ProbLog have clear and fixed semantics (the distribution semantics) that cannot be changed. These limitations have motivated the development of algebraic logical languages such as Dyna [19, 18] and aProbLog [34]. While standard probabilistic programming languages such as PRISM and ProbLog label facts with probabilities, Dyna and aProbLog use algebraic labels belonging to a semiring, which allows the use of other algebraic structures than the probabilistic semi-ring on top of the underlying logic programs. Dyna has been used to encode many AI problems, particularly in the area of natural language processing.

But so far, the expressiveness of these languages is still limited, which explains why many contemporary machine learning techniques involving probabilistic models, kernels and support-vector machines or neural networks cannot yet be modeled in these languages. Although Dyna and aProbLog have already been used to represent probabilistic models[1], and the Dyna papers mention some simple neural networks, there is — to the best of our knowledge — not yet work on using such languages for kernel-based learning. It is precisely this gap that we want to fill in this paper.

The key contribution of this paper is the introduction of kProbLog, an algebraic extension of Prolog, which can be used to declaratively model a wide range of problems and components from contemporary machine learning. More specifically, we shall show that kProbLog enables the declarative specification of four types of models that are central to machine learning today:

1. *tensor-based operations*: kProbLog allows to encode tensor operations in a way that is reminiscent of tensor relational algebra [33]. kProbLog supports recursion and is therefore more expressive than tensor relational algebra and related representations that have been proposed for relational learning [45].
2. *a wide family of kernel functions*: Declarative programming of kernels on structured data can be achieved via algebraic labels in the semiring of polynomials. Polynomials were previously used in combination with logic programming for sensitivity analysis by Kimmig et al (2011) and for data provenance by Green et al (2007). In this paper, we show that polynomials as kProbLog's algebraic labels enable the specification of label propagation and feature extraction schemas as those used in recent graph kernels such as Weisfeiler-Lehman (WL) graph kernels [53], propagation kernels [44] and graph kernels with continuous attributes such as graph invariant kernels [47]. Other graph kernels such as those based on random walks [31, 41] can be also easily declared in our language.
3. *probabilistic programs*: kProbLog is, as we show in Section 6, a generalization of the ProbLog probabilistic programming language.
4. *algorithmic differentiation*: kProbLog supports algorithmic differentiation by means of dual numbers [16]. Many learning strategies (ranging from collaborative filtering to neural networks and deep learning) that combine tensor-based

[1] Dyna does not handle the disjoint-sum problem; a more detailed explanation about reasoning about possible worlds and the disjoint-sum can be found in Section 6.

operations with gradient descent parameter tuning can therefore be implemented within the language.

The ability to define tensors, kernels, probabilistic models and support algorithmic differentiation are essential to contemporary machine learning. By supporting declarative modeling of such techniques in a relational setting, kProbLog contributes towards briding the gap between logical and relational learning and contemporary machine learning. We also provide an implementation of kProbLog and show using a number of experiments that it can be applied in practice, especially for prototyping.

At the more technical level, the key novelty of kProbLog as compared to Dyna and aProbLog is the introduction of two simple yet powerful mechanisms: the coexistence of multiple semirings within the same program, and the use of meta-functions for combining and manipulating algebraic values beyond simple "sum" and "product" operations. This allows to use kProbLog for declaratively specifying not only the logical component but also the algebraic one. The underlying idea being that the logic captures the structural aspect of the problem while the atom labels capture the algebraic aspect (including counts of substructures). We shall formally define the underlying semantics, provide an implementation of the language and prove its convergence properties.

The paper is organized as follows. First, we provide some basic notion of algebra and logic programming in Section 2. We then introduce kProbLog in Section 3, first giving a simplified version of the language based on a single semiring and then describing the full kProbLog language with multiple semirings and meta-functions. Section 3 also illustrates the relationship with tensor algebra. In Section 4 we then explain how kProbLog can be used to declaratively specify some complex state-of-the-art graph kernels, Section 5 shows that it is possible to perform algorithmic differentiation in kProbLog, while Section 6 shows that kProbLog is a proper generalization of ProbLog and hence, can be used as a probabilistic programming language. The work on kProbLog is then evaluated in Section 7: we show that kProbLog is expressive enough to allow for encoding kernels for some real world application domains and that the implementation is usable in that we obtain good statistical performance and runtimes on some benchmarks. Finally, in Section 8, we offer a comparative analysis of kProbLog and related languages, and draw some conclusions in Section 9.

2 Background

In this section, we provide some basic notions about algebra and logic programming.

2.1 Algebra

We now review some mathematical definitions.

Definition 1 A *monoid* is an algebraic structure $(\mathbb{S}, \cdot, e)$, where $\mathbb{S}$ is a set and $\cdot : \mathbb{S} \times \mathbb{S} \to \mathbb{S}$ is a binary operation, $e \in \mathbb{S}$ is the neutral element and the following properties are satisfied:

1. *associativity* $\forall a, b, c \in \mathbb{S}\ (a \cdot b) \cdot c = a \cdot (b \cdot c)$.
2. *neutral element* $\exists e : \forall s \in \mathbb{S} :\ e \cdot a = a \cdot e = a$.

A monoid is called *commutative* if $\forall a, b \in \mathbb{S} : a \cdot b = b \cdot a$.

Definition 2 A *semiring* is an algebraic structure $S = (\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ which satisfies the following properties:

1. $(\mathbb{S}, \oplus, 0_S)$ is a commutative monoid,
2. $(\mathbb{S}, \otimes, 1_S)$ is a monoid,
3. *distributive* multiplication left and right distributes over addition i.e. $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$.
4. *annihilating element* the neutral element of the sum 0_s is the annihilating element of multiplication: $0_s \otimes a = a \otimes 0_s = 0_s$.

A semiring is *commutative* if $\forall a, b \in \mathbb{S}\ a \otimes b = b \otimes a$ (i.e. $(\mathbb{S}, \otimes, 1_S)$ is a commutative monoid).

Definition 3 A semiring $S = (\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ is *complete* if it is possible to define sums for all families $(a_i | i \in I)$ of elements of $\mathbb{S}$ where I is an arbitrary index set, such that the following conditions are satisfied [15]:

1. $\bigoplus_{i \in \emptyset} a_i = 0_S$, $\bigoplus_{i \in \{j\}} a_i = a_j$, $\bigoplus_{i \in \{j,k\}} a_i = a_j \oplus a_k$ for $j \neq k$.
2. $\bigoplus_{j \in J} (\bigoplus_{i \in I_j} a_i) = \bigoplus_{i \in I} a_i$ for $\bigcup_{j \in J} I_j = I$ and $I_j \cap I_k = \emptyset, j \neq k$.
3. $\bigoplus_{i \in I} (c \otimes a_i) = c \otimes \left(\bigoplus_{i \in I} a_i\right), \quad \bigoplus_{i \in I} (a_i \otimes c) = \left(\bigoplus_{i \in I} a_i\right) \otimes c$.

These properties of a complete semiring S define *infinite sums* that extend finite sums, are associative and commutative, and satisfy the distributive law [15].

Definition 4 A semiring $(\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ is *naturally ordered* if the set $\mathbb{S}$ is partially ordered by the relation $\sqsubseteq$ such that $\forall a, b \in \mathbb{S} :\ a \sqsubseteq b$ if $\exists c \in \mathbb{S} : a \oplus c = b$. The partial order relation $\sqsubseteq$ on A is called *natural order* [38].

Definition 5 A semiring $(\mathbb{S}, \oplus, \otimes, 0_S, 1_S)$ is *ω-continuous* when: a) is complete b) is naturally ordered c) if $\bigoplus_{i=1}^{n} a_i \sqsubseteq c\ \forall n \in \mathbb{N}$ then $\bigoplus_{i \in \mathbb{N}} a_i \sqsubseteq c$ for all sequences $\{a_n\}_{i \in \mathbb{N}}$ in $\mathbb{S}$ and $c \in \mathbb{S}$.

2.2 Logic programming

A *term* `t` is recursively defined as a constant `c`, a logical variable `X` or a functor `f` applied on terms `t_i`, yielding `f(t_1,...,t_n)`. An atom takes the form `p(t_1,...,t_m)` where `p` is a predicate of arity `m` and `t_1,...,t_n` are terms. A *definite clause* `h :- b_1,...,b_n` is a universally quantified expression where `b_1`, ..., `b_n` and `h` are atoms. The atom `h` is called head of the clause while `b_1,...,b_n` is called body. The head `h` of the clause is true whenever all the atoms `b_1,...,b_n` in its body are true. A *fact* is a clause `h :- true` whose body is `true` and can be compactly written as `h`. A *definite clause program* P is a finite set of *definite clauses*, also called rules. An expression that does not contain variables is called *ground*. A *Herbrand base* A is the set of all the ground atoms that can be constructed from constants, functors and predicates in a definite clause program P. A Herbrand interpretation I of P is a truth value assignment to all

the atoms $a \in A$ and it is often written as the subset of true atoms. A Herbrand interpretation that satisfies all the rules in the program P is called *Herbrand model*. The model theoretic semantics of a definite clause program is given by its *least Herbrand model*, that is, the set of all ground atoms $a \in A$ that are entailed by the logic program P. Logical inference is the task of determining whether a *query* atom a, is entailed by a given logic program P. The two most common approaches to logical inference are backward reasoning and forward reasoning. The former starts from the query and reasons back tower the facts [46] and it is usually implemented in logic programming by SLD-resolution, while the latter starts from the facts and derives new true atoms using the immediate consequence operator T_P [20].

Definition 6 Let P be a ground definite clause program. The T_P-operator is a function that maps a Herbrand interpretation I to another Herbrand interpretation $T_P(I)$ as follows:

$$T_P(I) = \{\mathtt{h} | \mathtt{h} : -\mathtt{b_1}, ..., \mathtt{b_n} \in P \text{ and } \{\mathtt{b_1}, \ldots, \mathtt{b_n}\} \subseteq I\} \tag{1}$$

The least Herbrand model of a program P is the least fixed point of the T_P-operator, i.e. the least set of atoms I such that $T_P(I) = I$.

3 The kProbLog language

We introduce kProbLog in three different steps. In the first subsection, we assume that a single semiring is used; in the second subsection we introduce meta-functions and allow for multiple semirings; in the third, we present the inference algorithm of kProbLog, and analyze its convergence in the fourth subsection.

3.1 kProbLog$^{\mathbb{S}}$

kProbLog$^{\mathbb{S}}$ is an algebraic extension of Prolog with *labeled* facts and rules, where labels are chosen from a semiring S.

Definition 7 A kProbLog$^{\mathbb{S}}$ program P is a 4-tuple (F, R, S, ℓ) where:

- F is a finite set of facts;
- R is a finite set of definite clauses (also called rules);
- S is a semiring with sum $\oplus$ and product $\otimes$ operations; whose neutral elements are 0_S and 1_S respectively;
- $\ell : F \to \mathbb{S}$ is a function that maps facts to semiring values.

Definition 8 An algebraic interpretation $I_w = (I, w)$ of a ground kProbLog$^{\mathbb{S}}$ program P with facts F and atoms A is a set of tuples $(a, w(a))$ where a is an atom in the Herbrand base A and $w(a)$ is an algebraic formula over the fact labels $\{\ell(f) | f \in F\}$. We use the symbol $\emptyset$ to denote the empty algebraic interpretation, i.e. $\{(\text{true}, 1_S)\} \cup \{(a, 0_S) | a \in A\}$.

In this definition and below we adapt the notation of Vlasselaer et al (2015).

Definition 9 Let P be a ground algebraic logic program with algebraic facts F and Herbrand base A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the $T_{(P,S)}$-operator is $T_{(P,S)}(I_w) = \{(a, w'(a)) | a \in A\}$ where:

$$w'(a) = \begin{cases} \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\} \subseteq I \\ a:-b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) & \text{if } a \in A \setminus F \end{cases} . \tag{2}$$

Example 1 use of the algebraic T_P-operator.

kProbLog$^{\mathbb{S}}$	algebraic T_P-operator	numerical example	
`a :- a, b.`	$w(a) \otimes w(b)$	0.5×0.3	$w(a) = 0.5$
	$\oplus$	$+$	$w(b) = 0.3$
`a :- c.`	$w(c)$	0.9	$w(c) = 0.9$

The least fixed point can be computed using a semi-naive evaluation. When the semiring is non-commutative the product $\otimes$ of the weights $w(b_i)$ must be computed in the same order that they appear in the rule. kProbLog$^{\mathbb{S}}$ can represent matrices that in principle can have infinite size and can be indexed by using elements of the Herbrand universe of the program. We now show some elementary kProbLog$^{\mathbb{S}}$ programs that specify matrix operations:

	algebra	kProbLog$^{\mathbb{S}}$	numerical example

matrix A	A	`1::a(0, 0).`
		`2::a(0, 1).`
		`3::a(1, 1).`
			$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right]$

matrix B	B	`2::b(0, 0).`
		`1::b(0, 1).`
		`5::b(1, 0).`
		`1::b(1, 1).`
			$\left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right]$

matrix transpose	A^t	`c(I, J) :- a(J, I).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right]^t = \left[\begin{smallmatrix} 1 & 0 \\ 2 & 3 \end{smallmatrix}\right]$
matrix sum	$A + B$	`c(I, J) :- a(I, J).` `c(I, J) :- b(I, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] + \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 3 & 3 \\ 5 & 4 \end{smallmatrix}\right]$
matrix product	AB	`c(I, J) :-` `  a(I, K), b(K, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 12 & 3 \\ 15 & 3 \end{smallmatrix}\right]$
Hadamard product	$A \odot B$	`c(I, J) :-` `  a(I, J), b(I, J).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] \odot \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 2 & 2 \\ 0 & 3 \end{smallmatrix}\right]$
Kronecker product	$\text{kron}(A, B)$	`c(i(Ia, Ib), j(Ja, Jb)):-` `  a(Ia, Ja), b(Ib, Jb).`	$\left[\begin{smallmatrix} 1 & 2 \\ 0 & 3 \end{smallmatrix}\right] \otimes \left[\begin{smallmatrix} 2 & 1 \\ 5 & 1 \end{smallmatrix}\right] = \left[\begin{smallmatrix} 2 & 1 & 4 & 2 \\ 5 & 1 & 10 & 2 \\ 0 & 0 & 6 & 3 \\ 0 & 0 & 15 & 3 \end{smallmatrix}\right]$

The compound terms[2] `i/2` and `j/2`, were used to create the new indices that are needed by the Kronecker product. These definitions of matrix operations are reminiscent of tensor relational algebra [33]. Each of the above programs can be evaluated by applying the $T_{(P,S)}(I_w)$ operator only once. For each program we have a different definition of the C matrix that is represented by the predicate

[2] We use the notation *functor*/*arity* for compound terms.

`c/2`. As a consequence of Equation 2 all the algebraic labels of the `c/2` facts are polynomials in the algebraic labels of the `a/2` and `b/2` facts. We draw an analogy between the representation of a sparse tensor in coordinate format and the representation of an algebraic interpretation. A ground fact can be regarded as a tuple of indices/domain elements that uniquely identifies the cell of a tensor, the algebraic label of the fact represents the value stored in the cell.

Definition 10 An algebraic interpretation $I_w = (I, w)$ is the fixed point of the $T_{(P,S)}(I_w)$-operator if and only if for all $a \in A$, $w(a) \equiv w'(a)$, where $w(a)$ and $w'(a)$ are algebraic formulae for a in I_w and $T_{(P,S)}(I_w)$ respectively.

We denote with $T^i_{(P,S)}$ the function composition of $T_{(P,S)}$ with itself i times.

Proposition 1 (application of Kleene's theorem) *If S is an ω-continuous semiring the algebraic system of fixed-point equations $I_w = T_{(P,S)}(I_w)$ admits a unique least solution $T^\infty_{(P,S)}(\emptyset)$ with respect to the partial order $\sqsubseteq$ and $T^\infty_{(P,S)}(\emptyset)$ is the supremum of the sequence $T^1_{(P,S)}(\emptyset), T^2_{(P,S)}(\emptyset), \ldots, T^i_{(P,S)}(\emptyset)$. So $T^\infty_{(P,S)}(\emptyset)$ can be approximated by computing successive elements of the sequence. If the semiring satisfies the* ascending chain property *(see [21]) then $T^\infty_{(P,S)}(\emptyset) = T^i_{(P,S)}(\emptyset)$ for some $i \geq 0$ and $T^\infty_{(P,S)}(\emptyset)$ can be computed exactly [21].*

Examples of ω-continuous semirings are the Boolean semiring $(\{\mathrm{T}, \mathrm{F}\}, \vee, \wedge, \mathrm{F}, \mathrm{T})$, the tropical semiring $(\mathbb{N} \cup \{\infty\}, \min, +, \infty, 0)$ and the fuzzy semiring $([0, 1], \max, \min, 0, 1)$ [29].

Example 2 In this example, we show that some classic graph-theoretical algorithms can be described very concisely in kProbLog$^{\mathbb{S}}$. For this purpose, let us consider the following program:

```
1::edge(a, b).
3::edge(b, c).
7::edge(a, c).

path(X, Y):-
  edge(X, Y).
path(X, Y):-
  edge(X, Z), path(Z, Y).
```

Assuming that S is the Boolean semiring and that all the algebraic labels that are different from 0_S correspond to `true`$\in \mathbb{S}$, we obtain the Warshall algorithm for the transitive closure of a binary relation. If S is the tropical semiring, we obtain a specification of the Floyd-Warshall algorithm for all-pair shortest paths on graphs.

3.2 kProbLog

kProbLog generalizes kProbLog$^{\mathbb{S}}$ in two ways: it allows *multiple semirings* to coexist in the same program, and it enriches the algebraic expressivity by means of *meta-functions* and *meta-clauses*.

Every algebraic predicate in a kProbLog program needs to be associated with its own semiring S via the built-in predicate `declare(P, S)` where `P` is either a predicate (written in the form *name/arity*) or a list of predicates and `S` specifies a member of the kProbLog semiring library[3]. For example, the directive

[3] The library contains can be extended with the Python language.

```
:- declare(vertex/2, polynomial(real)).
```

is used to associate `vertex/2` with the semiring of polynomials over real numbers.

Definition 11 (meta-function) A meta-function `m`: $\mathbb{S}_1 \times \ldots \times \mathbb{S}_k \mapsto \mathbb{S}'$ is a function that maps k semiring values $x_i \in \mathbb{S}_i,\ i = 1, \ldots, k$ to a value of type $\mathbb{S}'$, where $\mathbb{S}_1, \ldots, \mathbb{S}_k$ and $\mathbb{S}'$ can be distinct sets. If `a_1,...,a_k` are algebraic atoms, in kProbLog we use the syntax `@m[a_1,...,a_k]` to express the application of meta-function `@m` to the values $w($`a_1`$), ..., w($`a_k`$)$ of the atoms `a_1,...,a_k`.

Definition 12 (meta-clause) A meta-clause `h :- b_1,...,b_n` is a universally quantified expression where `h` is an atom and `b_1,...,b_n` can be either atoms or meta-functions applied to other algebraic atoms. The head predicate of a meta-clause, the algebraic atoms in the body, and the return types of the meta-functions in the body must all belong to the same semiring.

The introduction of meta-functions in kProbLog allows us to deal with other algebraic structures such as rings that require the additive inverse `@minus/1` and fields that require the additive inverse and the multiplicative inverse `@inv/1`.

Definition 13 (kProbLog program) A kProbLog program P is a union of kProbLog$^{\mathbb{S}_i}$ programs and meta-clauses.

3.2.1 kProbLog T_P-operator with meta-functions

The algebraic T_P-operator of kProbLog is defined on the meta-transformed program.

Definition 14 (meta-transformed program) A meta-transformed kProbLog program is a kProbLog program in which all the meta-functions are expanded to algebraic atoms. For each rule `h :- b_1,...,@m[a_1,...,a_k],...,b_n` in the program P each meta-function `@m[a_1,...,a_k]` is replaced by an atom `b'` and a meta-clause `b':-@m[a_1,...,a_k]` is added to the program P.

Definition 15 (algebraic T_P-operator with meta-functions) Let P be a meta-transformed kProbLog program with facts F and atoms A. Let $I_w = (I, w)$ be an algebraic interpretation with pairs $(a, w(a))$. Then the T_P-operator is $T_P(I_w) = \{(a, w'(a)) | a \in A\}$ where:

$$w'(a) = \begin{cases} \ell(a) & \text{if } a \in F \\ \bigoplus\limits_{\substack{\{b_1,\ldots,b_n\} \subseteq I \\ a:-b_1,\ldots,b_n}} \bigotimes\limits_{i=1}^{n} w(b_i) \oplus \bigoplus\limits_{\substack{\{b_1,\ldots,b_k\} \subseteq I \\ a:-@m[b_1,\ldots,b_k]}} m(w(b_1), \ldots, w(b_k)) & \text{if } a \in A \setminus F\,. \end{cases} \quad (3)$$

Example 3 of the algebraic T_P-operator with meta-functions.

kProbLog	algebraic T_P-operator	numerical example	
`a :- a, b.`	$w(a) \otimes w(b)$	0.5×0.3	$w(a) = 0.5$
	$\oplus$	$+$	$w(b) = 0.3$
`a :- @sin[c].`	$sin(w(c))$	$0.78...$	$w(c) = 0.9$

Where we used the identity $\sin(0.9) = 0.78\ldots$

3.2.2 Recursive kProbLog program with meta-functions

Recursion is a basic tool in logic programming. For our purposes, it is necessary in most useful computations on structured data such as shortest paths (see Example 2), or random walk graph kernels (See Section 4.4.3). Weights need to be updated whenever the groundings of a predicate appear in the cycles of the ground program.

Definition 16 A ground program P is cyclic if it contains a cycle. A cycle is a sequence of rules $r_1, \ldots, r_n$ such that the head of r_i is contained in the body of r_{i-1} for $i = 2, \ldots, m$ and the head of r_1 is contained in r_m. A ground rule that is contained in a cycle is called *cyclic rule*, otherwise it is called *acyclic rule*

kProbLog allows both additive and destructive updates, as specified by the built-in predicate `declare(P, S, U)` where `U` can be either `additive` or `destructive`.

Definition 17 Additive and destructive updates.

If $r_1, \ldots, r_n$ are all ground cyclic rules with head `h`, the value of the weight update value $\Delta w(\mathtt{h})$ is computed as:

$$\Delta w(\mathtt{h}) = \bigoplus_{i=1}^{n} T_P(r_i). \tag{4}$$

According to the declaration of the predicate of atom `h` the update will be either

- `additive` $w(\mathtt{h}) = w(\mathtt{h}) \oplus \Delta w(\mathtt{h})$ or
- `destructive` $w(\mathtt{h}) = \Delta w(\mathtt{h})$.

The distinction between `additive` and `destructive` is only relevant for cyclic rules. In Section 3.3 we give the evaluation algorithm of kProbLog which uses this kind of update when necessary.

Programs such as the transitive closure of a binary relation (see Example 2) or the compilation of ProbLog programs with sentential decision diagrams (SDD) [8] require additive updates (see Section 6). Destructive updates are necessary to specify iterated function composition, as shown in the next example.

Example 4 Suppose we wish to compute

$$\lim_{n \to +\infty} g^n(x_0)$$

where $g \equiv x(1-x)$ and

$$g^n \equiv \begin{cases} g \circ g^{n-1} & \text{if } n>0 \\ g & \text{if } n=0. \end{cases} \tag{5}$$

The value $g^n(x_0)$ can be obtained in kProbLog as follows:

```
:- declare(x, real, destructive).
:- declare(x0, real).
0.5::x0.
x :- x0.
x :- @g[x].
```

The above program has the following behavior: the weight $w(x)$ of `x` is initialized to $w(x_0) = 0.5$ and then updated at each step according to the rule $w'(x) = g(w(x))$ (destructive update). An additive update $w'(x) = w(x) + g(w(x))$ would have produced an incorrect result in this case.

3.2.3 The Jacobi method

We already showed that kProbLog can express linear algebra operations. We now combine recursion and meta-functions in an algebraic program that specifies the Jacobi method [28], an iterative algorithm used for solving diagonally dominant systems of linear equations $A\mathbf{x} = \mathbf{b}$.

We consider the field of real numbers $\mathbb{R}$ (i.e. kProbLog$^{\mathbb{R}}$) as semiring together with the meta-functions `@minus` and `@inv` that provide the inverse element of sum and product respectively.

The A matrix must be split according to the Jacobi method:

$D = \text{diag}(A)$ `d(I, I) :- a(I, I).`

$R = A - D$ `r(I, J) :- a(I, J), I \= J.`

The solution $\mathbf{x}^*$ of $A\mathbf{x} = \mathbf{b}$ is computed iteratively by finding the fixed point of $\mathbf{x} = D^{-1}(\mathbf{b} - R\mathbf{x})$. We call E the inverse of D. Since D is diagonal also E is a diagonal matrix:

$e_{ii} = \texttt{inv}(d_{ii}) = \frac{1}{d_{ii}}$ `e(I, I) :- @inv[d(I, I)].`

and the iterative step can be rewritten as $\mathbf{x} = E(\mathbf{b} - R\mathbf{x})$.

Making the summations explicit we can write:

$$x_i = \sum_k e_{ik} \left(b_k - \sum_l r_{kl} x_l \right) \tag{6}$$

then we can extrapolate the term $\sum_l r_{kl} x_l$ turning it into the aux_k definition:

$x_i = \sum_k e_{ik} \left(b_k - aux_k \right)$

$aux_k = \sum_l r_{kl} x_l$

```
:- declare(x/1, real, destructive).
:- declare(aux/1, real, destructive).
x(I) :-
  e(I, K), @subtraction[b(K), aux(K)].

aux(K) :-
  r(K, L),  x(L).
```

where `@subtraction/2` represents the subtraction between real numbers, `x/1` and `aux/1` are mutually recursive predicates. Because `x/1` needs to be initialized (perhaps at random) we also need the clause:

$x_i = init_i$ `x(I) :- init(I).`

where `init/1` is a unary predicate. This example also shows that kProbLog is more expressive than tensor relational algebra because it supports recursion.

The introduction of meta-functions makes the result of the evaluation of a kProbLog program dependent on the order in which rules and meta-clauses are evaluated. For this reason we explain the order adopted by the kProbLog language.

3.3 kProbLog implementation

Pseudo-code for the interpreter is given in Algorithm 1. A kProbLog program P is first grounded to a kProbLog program by the procedure Ground and is then evaluated by partitioning Ground(P) into a topologically ordered sequence of strata $P_1, \ldots, P_n$ such that

- every stratum P_i is a set of ground atoms which is both is maximal and strongly connected (i.e. each ground atom in P_i depends on every other ground atom in P_i);
- a ground atom in an acyclic stratum P_i can only depend[4] on the ground atoms from the previous strata $\bigcup_{j<i} P_j$;
- an ground atom in a cyclic stratum can depend on the ground atoms in $\bigcup_{j\leq i} P_j$.

Program evaluation starts by initializing the weight $w(a)$ of each ground atom $a \in \textsc{Ground}(P)$ to 0_S, where S is the semiring of the atom. The strata are then visited in topological order and the weights are updated as follows: if the stratum P_i is acyclic, then the algebraic T_P-operator is applied only once per atom; if P_i is cyclic then the algebraic T_P-operator is first applied to the acyclic rules and meta-clauses and then, repeatedly until convergence, to the cyclic rules and meta-clauses. The procedure TpOperator takes as input a *rule* and the atom weights w and returns an algebraic value derived from the application of the algebraic T_P-operator.

The update for a weight $w(a)$ of a cyclic atom a is computed by accumulating the result of the application of the T_P-operator to all the cyclic rules with head a. The new weight is then computed as $w(a) = w(a) + \Delta w(a)$ (additive updates) or $w(a) = \Delta w(a)$ (destructive updates).

If P_i is a cyclic stratum, then it is the responsibility of the programmer to ensure convergence of the algebraic T_P-operator. Nevertheless if the P_i is a cyclic stratum in which only rules are cyclic then all the atoms in P_i are on the same semiring[5] S and so P_i has the same convergence properties of a kProbLog$^{\mathbb{S}}$ program (see Theorem 2 on page 14). Whenever we apply the algebraic T_P-operator we use the Jacobi evaluation. Jacobi and Gauss-Seidel evaluations are two alternatives to perform naive evaluation of Datalog programs and are also well known in numerical analysis [5]. We choose Jacobi over Gauss-Seidel evaluation because the former produces side effects on the algebraic weighs w only after (and not during) the computation of the algebraic T_P-operator. In this way the execution of the program is not affected by the order in which rules and meta-clauses are evaluated.

This program evaluation procedure is an adaptation the work of Whaley et al (2005) on Datalog and binary decision diagrams. kProbLog was implemented in Python 3.5 using Gringo 4.5[6] as grounder. The source code of our kProbLog implementation is available at `https://github.com/orsinif/kProbLogDSL`.

Example 4 (continued) Evaluation of a cyclic program. The cyclic program P in Section 3.2.2 is already ground and contains two ground atoms `x0` and `x`. The ground atoms `x0` and `x` correspond to two nodes in the dependency graph, while `x0` is a fact and does not have incoming arcs, `x` has two dependencies/incoming arcs which are `x0` and itself. As shown in Figure 1 P is then subdivided in two strata P_1 and P_2: P_1 contains `x0` and is acyclic, P_2 contains `x` and is cyclic.

The algebraic T_P-operator is applied only once for acyclic rules and multiple times, until convergence, for cyclic rules (i.e. `x:- @g[x].`)

[4] We say that an atom a *directly depends* on an atom b if a is the head of a rule or a meta-clause and b is a body literal or an argument of a meta-function in the meta clause. We say that an atom a *depends* on an atom b either if a directly depends on b or there is an atom c such that a directly depends on c and c depends on b.

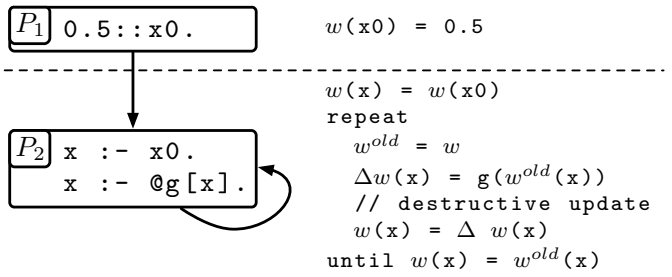
[5] Atoms of distinct semirings cannot be mutually dependent without using meta-clauses.

[6] `https://potassco.org`

Algorithm 1 Pseudo-code for the kProbLog evaluation procedure.

```
kProbLog(P)
 1  F = Facts(P)
 2  for f ∈ F
 3       w(f) = ℓ(f)
 4  [P_1, ..., P_n] = TopSort(GetStrata(Ground(P)))
 5  for i = 1, ..., n
 6       for a ∈ P_i \ F
 7            w(a) = 0_s
 8       w_old = w
 9       ACYCLIC = AcyclicRules(P_i)
10       CYCLIC = CyclicRules(P_i)
11       for rule ∈ ACYCLIC
12            h = Head(rule)
13            w(h) = w_old(h) ⊕ TpOperator(rule, w_old)
14       repeat
15            w_old = w
16            for rule ∈ CYCLIC
17                 Δw(Head(rule)) = 0_s
18            for rule ∈ CYCLIC
19                 h = Head(rule)
20                 Δw(h) = Δw(h) ⊕ TpOperator(rule, w_old)
21            for rule ∈ CYCLIC
22                 if rule is additive
23                      w(Head(rule)) = w_old(Head(rule)) ⊕ Δw(Head(rule))
24                 else          // rule is destructive
25                      w(Head(rule)) = Δw(Head(rule))
26       until w_old = w
```

Fig. 1 Cyclic program of Example 4: dependency graph with stratification and corresponding weight updates.



3.4 Convergence analysis of the kProbLog interpreter

In order to analyze the convergence of the kProbLog interpreter on a kProbLog program P, we assume that all the meta-functions in P terminate and that the *finite support* condition [51] holds.

The *finite support* condition is commonly used in probabilistic logic programming and ensures that the Ground procedure outputs a finite ground program.

The convergence properties of kProbLog are characterized by the following theorems.

Theorem 1 (Convergence of acyclic kProbLog programs) *The evaluation of an acyclic kProbLog program P invokes the algebraic T_P-operator exactly once for each ground rule in* $\textsc{Ground}(P)$ *and terminates.*

Theorem 2 (Convergence of kProbLog programs) *The evaluation of a kProbLog program is guaranteed to terminate only if all the cyclic strata are kProbLog$^{\mathbb{S}_i}$ programs where $\mathbb{S}_i$ are ω-continuous semirings.*

The proofs of Theorems 1 and 2 are reported in Appendix Section A.

Theorem 1 can be used to prove the convergence of the elementary programs that specify matrix operations in Section 3.1 and the convergence of the WL algorithm that we shall se in Section 4.2. Theorem 2 ensures the convergence of the cyclic program in Example 2 when an ω-continuous semiring is used for the algebraic labels, but not the convergence of the program in Example 4. While the cyclic program in Example 4 actually converges, this property cannot be entailed from Theorem 2. Indeed the program in Example 4 has a cyclic stratum P_2 (see Figure 1) involving a meta-function (i.e. `@g`). Stratum P_2 is not a kProbLog$^{\mathbb{S}}$ program because kProbLog$^{\mathbb{S}}$ programs do not admit meta-functions and for this reason we cannot apply Theorem 2 to Example 4.

kProbLog programs whose strata are either acyclic programs or cyclic programs on ω-continuous semirings are guaranteed to converge and it is possible to verify that these conditions are met at runtime. However, since kProbLog is an extension of Prolog, which is a Turing-complete language we choose to allow meta-functions and non-ω-continuous semirings in cyclic strata. In this way we do not restrict the expressivity of the language.

4 Kernel programming

We now show that kProbLog can be used to declaratively encode state-of-the-art graph kernels. But before doing so, we introduce the semiring $\mathbb{S}[\mathbf{x}]$ that can be used for feature extraction.

4.1 kProbLog$^{\mathbb{S}[\mathbf{x}]}$: polynomials for feature extraction

kProbLog$^{\mathbb{S}[\mathbf{x}]}$ labels facts and rule heads with polynomials over the semiring S. kProbLog$^{\mathbb{S}[\mathbf{x}]}$ is a particular case of kProbLog$^{\mathbb{S}}$ because polynomials over semirings are semirings in which addition and multiplication are defined as usual.

While polynomials have been used in combination with logic programming for provenance [29] and sensitivity analysis [34], we use multivariate polynomials to represent the explicit feature map of a graph kernel.

Definition 18 (Multivariate polynomials over commutative semirings) A multivariate polynomial $\mathcal{P} \in S[\mathbf{x}]$ can be expressed as:

$$\mathcal{P}(\mathbf{x}) = \bigoplus_{i=1}^{n} c_i \mathbf{x}^{\mathbf{e}_i} = \bigoplus_{i=1}^{n} c_i \otimes \bigotimes_{t \in T} x_t^{e_{it}} \tag{7}$$

where $c_i \in \mathbb{S}$ are the coefficients of the i^{th} monomial and $\mathbf{x}$, $\mathbf{e}$ are vectors of variables and exponents respectively. The vector $\mathbf{x}$ is indexed by ground terms $t \in T$.

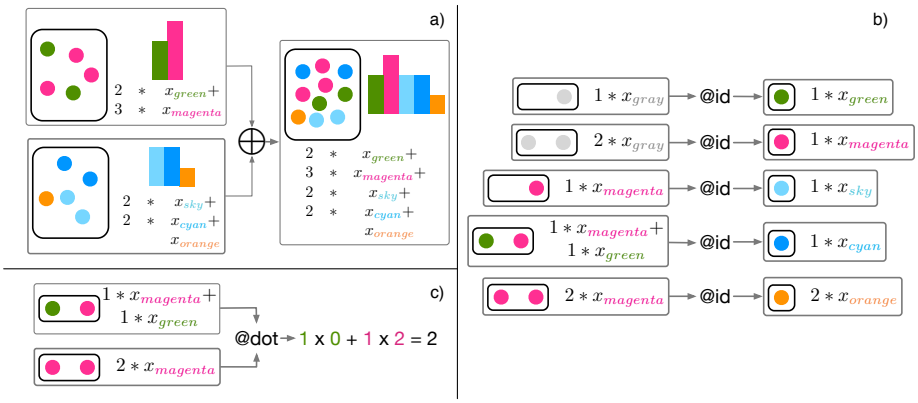


Fig. 2 Using polynomials for representing multisets: a) multiset union corresponds to sum over polynomials; b) the inner product (kernel) between multisets corresponds to product over polynomials; c) multiset compression via the `@id` meta-function over polynomials.

WL, propagation and neighborhood-subgraph-pairwise-distance kernel features can all be cast into this representation. These graph kernels propagate messages through the structure of the graphs, these messages can be represented as multisets of terms (elements of the Herbrand universe). Indeed we can represent a multiset μ of terms as a polynomial:

$$\mathcal{P}_\mu(\mathbf{x}) = \sum_{t \in \mu} \sharp t \cdot x_t \tag{8}$$

where $\sharp$ counts the number of occurrences of the terms in the multiset μ.

4.1.1 Operations for feature extraction

Sum of polynomials The semiring sum $\oplus$ between polynomials is used in kProbLog$^{\mathbb{S}[\mathbf{x}]}$ to sum features or equivalently compute a multiset union operation (see Figure 2.a).

Inner product between polynomials The kProbLog `@dot` meta-function corresponds to the inner product on multivariate polynomials over $S[\mathbf{x}]$:

$$\langle \mathcal{P}(\mathbf{x}), \mathcal{Q}(\mathbf{x}) \rangle = \bigoplus_{\substack{(p,\mathbf{e}) \in \mathcal{P} \\ (q,\mathbf{e}) \in \mathcal{Q}}} p \otimes q. \tag{9}$$

For each monomial (uniquely identified by the vector of exponents $\mathbf{e}$) that appears in both the polynomials $\mathcal{P}$ and $\mathcal{Q}$, Equation 9 computes the product between their coefficients p and q respectively. These products are then summed together to obtain the value of the inner product. Natural choices for the semiring are polynomials over integers $\mathbb{Z}[\mathbf{x}]$ and real numbers $\mathbb{R}[\mathbf{x}]$, these semirings also ensure that the inner-product is positive semidefinite.

Example 5 The following multivariate polynomials over integers:

$$\begin{aligned}\mathcal{P}(x_1, x_2, x_3) &= 2x_1 + 3x_1x_2 + x_2x_3^2 \\ \mathcal{Q}(x_1, x_2, x_3) &= 4x_1 + 3x_1x_3 + 3x_2x_3^2\end{aligned} \tag{10}$$

can be expressed as two sets of coefficient-exponent pairs $\mathcal{P} = \{(2, [1, 0, 0]), (3, [1, 1, 0]), (1, [0, 1, 2])\}$ and $\mathcal{Q} = \{(4, [1, 0, 0]), (3, [1, 0, 1]), (3, [0, 1, 2])\}$ respectively. The two polynomials have the vectors of exponents $[1, 0, 0]$ and $[0, 1, 2]$ in common, each contributes to the inner product by $2 \times 4 = 8$ and $1 \times 3 = 3$ respectively. The value of the inner product between $\mathcal{P}(x_1, x_2, x_3)$ and $\mathcal{Q}(x_1, x_2, x_3)$ is the sum of such contributions $8 + 3 = 11$.

In kProbLog, the meta-function `@dot/2`. computes the inner product between two algebraic atoms $\mathcal{P}(\mathbf{x})$`::a` and $\mathcal{Q}(\mathbf{x})$`::b`. An example is shown in Figure 2.b where the multisets of terms {{`green`, `magenta`}} and {{`magenta`, `magenta`}} are represented by the following two polynomials:

$$\begin{aligned}\mathcal{P}(x_{green}, x_{magenta}) &= x_{green} + x_{magenta} \\ \mathcal{Q}(x_{green}, x_{magenta}) &= 2x_{magenta}\end{aligned} \tag{11}$$

Another useful meta-function in the context of kernel design is `@rbf/3`. It takes as input an atom labeled by a non-negative real value γ and two atoms labeled with the polynomials $\mathcal{P}$ and $\mathcal{Q}$ and it computes the radial basis function kernel $exp\{-\gamma\|\mathcal{P} - \mathcal{Q}\|^2\}$, where $\|\mathcal{P} - \mathcal{Q}\|^2 = \langle\mathcal{P}, \mathcal{P}\rangle + \langle\mathcal{Q}, \mathcal{Q}\rangle - 2\langle\mathcal{P}, \mathcal{Q}\rangle$.

The compression meta-function The `@id/1` meta-function `@id`: $\mathbb{S}[\mathbf{x}] \to \mathbb{S}[\mathbf{x}]$ is injective. `@id/1` transforms a polynomial $\mathcal{P}(\mathbf{x})$ to a new term t and returns the polynomial `@id`$[\mathcal{P}(\mathbf{x})] = 1.0 \cdot x(t)$. This function can be used to compress a multivariate polynomial to a new polynomial in a single variable. We use the `@id` meta-function for polynomial compression as Shervashidze et al (2011) use the function f to compress multisets of labels. We now show how these functions are used to specify graph kernels.

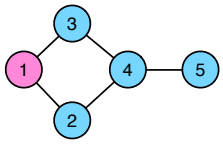
4.2 The Weisfeiler-Lehman algorithm

The one-dimensional WL method is an iterative vertex classification algorithm for the graph isomorphism problem. It begins by coloring vertices with their labels and, at each round, it recolors vertices by a "compressed" version of the multiset of colors at their neighbors. If, at any iteration, two graphs have different sets of vertex colors they cannot be isomorphic. We will use polynomials to represent WL colors, associating variables with colors and using integer coefficients to encode the number of occurrences of a color in a multiset.

A colored graph $G = (V, E, \ell)$, where V is a set of vertices, $E \subseteq V \times V$ is the set of the edges, and $\ell : V \mapsto \Sigma$ is a function that maps vertices to a color alphabet Σ, can be declared in kProbLog as follows:

```
:- declare(vertex/2, polynomial(int)).
:- declare(edge_asymm/3, boolean).
:- declare(edge/3, polynomial(int)).

1 * x(pink)::vertex(graph_a, 1).
1 * x(blue)::vertex(graph_a, 2).
1 * x(blue)::vertex(graph_a, 3).
1 * x(blue)::vertex(graph_a, 4).
1 * x(blue)::vertex(graph_a, 5).

edge_asymm(graph_a, 1, 2).
edge_asymm(graph_a, 1, 3).
edge_asymm(graph_a, 2, 4).
edge_asymm(graph_a, 3, 4).
edge_asymm(graph_a, 4, 5).
```

```
1.0::edge(Graph, A, B):-
    edge_asymm(Graph, A, B).

1.0::edge(Graph, A, B):-
    edge_asymm(Graph, B, A).
```



where the predicate `edge_asymm/3` is implicitly cast to type *integer* and then to type *polynomial over integers* when it appears in the definition of `edge/3`. The WL color of vertex v at iteration h can be written as:

$$\mathcal{L}^h(v) = \begin{cases} \ell(v) & \text{if } h = 0 \\ f(\mathcal{L}^{h-1}(v), \{\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}\}) & \text{if } h > 0 \end{cases} \tag{12}$$

where $\mathcal{N}(v)$ is the set of the vertex neighbors of v, $\{\{\mathcal{L}^{h-1}(w) | w \in \mathcal{N}(v)\}\}$ is the multiset of their colors at step $h-1$, and f is a variadic injective function that maps its arguments to a new color in Σ. Equation 12 can be expressed in kProbLog as shown below, where f is implemented via the `@id` meta-function:

```
:- declare(wl_color/3,
    polynomial(int)).
:- declare(wl_color_multiset/3,
    polynomial(int)).

wl_color_multiset(H, Graph, V):-
    edge(Graph, V, W),
    wl_color(H, Graph, W).
```

```
wl_color(0, Graph, V) :-
    vertex(Graph, V).

wl_color(H, Graph, V):-
    1 <= H, H <= MAX_ITER,
    @id[wl_color(H-1, Graph, V),
    wl_color_multiset(H-1, Graph, V)].
```

The WL algorithm has been specified as an acyclic program. Indeed, while `wl_color/3` and `wl_color_multiset/3` are mutually recursive `wl_color/3` at step `H` depends on `wl_color/3` and `wl_color_multiset/3` at step `H-1`, therefore is acyclic and we can apply Theorem 1 to verify that it converges.

4.3 Graph kernels

In this section we give the declarative specification of some recent graph kernels such as the WL graph kernel [53], propagation kernels [44] and graph invariant kernels [47]. These methods have been applied to different domains such as natural language processing [47], computer vision [44] and bioinformatics [53, 44, 47].

4.4 Weisfeiler-Lehman graph kernel and Propagation kernels

The WL graph kernel is defined using a base kernel [53] that computes the inner-product between the histograms of WL colors of two graphs `Graph` and `GraphPrime`.

$$\phi^{(h)}(G) = \sum_{v \in V(G)} \mathcal{P}_{\text{WL}}^{(h)}(v)$$

```
:- declare(phi/2, real).
phi(H, Graph):-
  wl_color(H, Graph, V).
```

$$k_{\text{BASE}}^{(h)}(G, G') = \langle \phi^{(h)}(G), \phi^{(h)}(G') \rangle$$

```
:- declare(base_kernel/3, real).
base_kernel(H, Graph, GraphPrime):-
  @dot[phi(H, Graph),
       phi(H, GraphPrime)].
```

Where $\mathcal{P}_{\text{WL}}^{(h)}(v)$ is the polynomial that represents the WL color of vertex v at step h.

The WL graph kernel [53] with `H` iterations is the sum of base kernels computed for consecutive WL labeling steps $1, \ldots,$`H` on the graphs `Graph` and `GraphPrime`:

$$k_{\text{WL}}^{(H)}(G, G') = \sum_{h=0}^{H} k_{\text{BASE}}^{(h)}(G, G')$$

```
:- declare(kernel_wl/3, real).
kernel_wl(0, Graph, GraphPrime):-
  base_kernel(0, Graph, GraphPrime).

kernel_wl(H, Graph, GraphPrime):-
  H > 0, H1 is H - 1,
  kernel_wl(H1, Graph, GraphPrime).

kernel_wl(H, Graph, GraphPrime):-
  H > 0,
  base_kernel(H, Graph, GraphPrime).
```

The above equation can be rewritten using a recursive definition which is closer to the kProbLog specification as follows

$$k_{\text{WL}}^{(H)}(G, G') = \begin{cases} k_{\text{BASE}}^{(0)}(G, G') & \text{if } H = 0 \\ k_{\text{WL}}^{(H-1)}(G, G') + k_{\text{BASE}}^{(H)}(G, G') & \text{if } H > 0. \end{cases} \tag{13}$$

Propagation kernels [44] extend the WL graph kernel and can adopt different label propagation schemas. Neumann et al (2012) implement propagation kernels using locality sensitive hashing (LSH). The kProbLog specification is almost identical to the one of the WL except that the `@id` meta-function is to be replaced with a meta-function that does LSH.

LSH discretizes vectors to integer identifiers so that vectors which are similar have the same integer identifier with high probability.

4.4.1 Shortest path Weisfeiler-Lehman graph kernel

The shortest path WL [53] graph kernel is a specialization of the WL graph kernel where the base kernel counts the number of common occurrences of triplets of the form (a, b, d) between two graphs G and G'. The triplet (a, b, d) represents the occurrence of two vertices v and w at distance d with colors $a = \mathcal{L}(v)$ and $b = \mathcal{L}(w)$. To compactly encode the shortest path variant of the WL graph kernel we begin by computing all-pairs shortest paths using the tropical semiring:

```
:- declare(distance/3, tropical).
distance(Graph, V, W):-
  edge(Graph, V, W).

distance(Graph, V, W):-
  distance(Graph, V, U), edge(Graph, U, W).
```

We then introduce predicate `triplet_id(Graph, H,V, W)` of type `polynomial (real)` that associates each pair of vertices `V` and `W` of graph `Graph` with their $\mathtt{H}^{\text{th}}$-iteration WL color, together with their shortest path distance, d, obtained by calling the `@id/1` meta-function on the distance predicate. Finally, triplet (a, b, d) is represented as the monomial $x_a x_b x_d$ via auxiliary predicate `triplet(Graph, H, V, W)` and compressed to a color by using again the `@id` meta-function:

```
:- declare(triplet/4, polynomial(real)).
:- declare(triplet_id/4, polynomial(real)).

triplet(Graph, H, V, W):-
  wl_color(H, Graph, V),
  wl_color(H, Graph, W),
  @id[distance(Graph, V, U)].

triplet_id(Graph, H, V, W):-
  @id[triplet_id(Graph, H, V, U)].
```

This specification fully employs the expressive power of kProbLog using meta-functions and two distinct semirings that encode distances and vertex colors (the base kernel for this variant of the WL graph kernel is obtained by replacing predicate `wl_color/3` defined in Section 4.4 with predicate `triplet_id/4` defined above).

4.4.2 Graph invariant kernels

Graph Invariant Kernels (GIKs, pronounce "geeks") are a recent framework for graph kernels with continuous attributes [47]. GIKs compute a similarity measure between graphs G and G' matching them at vertex level according to the formula:

$$k(G, G') = \sum_{v \in V(G)} \sum_{v' \in V(G')} w(v, v') k_{\text{ATTR}}(v, v') \tag{14}$$

where $w(v, v')$ is the structural weight matrix and $k_{\text{ATTR}}(v, v')$ is a kernel on the continuous attributes of the graphs. The kProbLog specification is parametrized by the logical variable `R`, which is needed for the definition of the structural weight matrix $w(v, v')$.

```
:- declare(gik_radius/3, real).
gik_radius(R, Graph, GraphPrime):-
  w_matrix(R, Graph, V, GraphPrime, VPrime),
  k_attr(Graph, V, GraphPrime, VPrime).
```

where `gik_radius/3`, `w_matrix/5` and `k_attr/4` are algebraic predicates on the real numbers semiring, which is represented with floats for implementation purposes. Assuming that we want to use the RBF with $\gamma = 0.5$ kernel on the vertex attributes we can write:

```
:- declare(rbf_gamma_const/0, real).
:- declare(k_attr/4, real).
0.5::rbf_gamma_const.
k_attr(Graph, V, GraphPrime, VPrime):-
  @rbf[rbf_gamma_const, attr(Graph, V), attr(GraphPrime, VPrime)].
```

where `attr/2` is an algebraic predicate that associates to the vertex `V` of a `Graph` a polynomial label. To associate to vertex `v_1` of `graph_a` the 4-dimensional feature $[1, 0, 0.5, 1.3]$ we would write:

```
:- declare(attr/2, polynomial(real)).
1.0 * x(1) + 0.5 * x(3) + 1.3 * x(4)::attr(graph_a, v_1).
```

while the meta-function `@rbf/3` takes as input an atom `rbf_gamma_const` labeled with the γ constant and the atoms relative to the vertex attributes.

The structural weight matrix $w(v, v')$ is defined as:

$$w(v,v') = \sum_{g \in \mathcal{R}^{-1}(G)} \sum_{g' \in \mathcal{R}^{-1}(G')} k_{\text{INV}}(v,v') \frac{\delta_m(g,g')}{|V_g||V_{g'}|} \mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}. \quad (15)$$

The weight $w(v, v')$ measures the structural similarity between vertices and is defined combining an $\mathcal{R}$-decomposition relation, a function $\delta_m(g, g')$ and a kernel on vertex invariants k_{INV} [47]. In our case the $\mathcal{R}$-decomposition generates `R`-neighborhood subgraphs (as those used in the experiments of Orsini et al (2015)).

There are multiple ways to instantiate GIKs, we choose the version called LWL_V, which can achieve very good accuracies most of the time as shown by Orsini et al (2015). LWL_V uses `R`-neighborhood subgraphs $\mathcal{R}$-decomposition relation, computes the kernel on vertex invariants $k_{\text{INV}}(v, v')$ at the pattern level (*local* GIK) and uses $\delta_m(g, g')$ to match subgraphs that have the same number of nodes.

A `R`-neighborhood subgraph of a graph G from a vertex v is the subgraph induced by all the vertices in G whose shortest-path distance from v is less than or equal to `R`.

In kProbLog we would write:

```
:- declare(w_matrix/5, real).
w_matrix(R, Graph, V, GraphPrime, VPrime):-
  vertex_in_ball(Graph, R, BallRoot, V),
  vertex_in_ball(GraphPrime, R, BallRootPrime, VPrime),
  delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime),
  @inv[ball_size(R, Graph, BallRoot)],
  @inv[ball_size(R, GraphPrime, BallRootPrime)],
  k_inv(Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime).
```

where:

a) `vertex_in_ball(R, Graph, BallRoot, V)` is a Boolean predicate which is true if `V` is a vertex of `Graph` inside a `R`-neighborhood subgraph rooted in `BallRoot`. `vertex_in_ball/4` encodes both the term $\mathbb{1}\{v \in V_g \wedge v' \in V_{g'}\}$ and the pattern generation of the decomposition relation $g \in \mathcal{R}^{-1}(G)$.

```
:- declare(vertex_in_ball/4, bool).
vertex_in_ball(0, Graph, Root, Root):-
  vertex(Graph, Root).

vertex_in_ball(R, Graph, Root, V):-
  R > 0, R1 is R - 1,
  vertex_in_ball(R1, Graph, Root, V).
```

```
vertex_in_ball(R, Graph, Root, V):-
  R > 0, R1 is R - 1,
  edge(Graph, Root, W),
  vertex_in_ball(R1, Graph, W, V).
```

b) `delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime)` matches subgraphs with the same number of vertices

```
:- declare(delta_match/5, real).
:- declare(v_id/3, polynomial(real)).
:- declare(ball_size/3, int).

delta_match(R, Graph, BallRoot, GraphPrime, BallRootPrime):-
  @eq[v_id(R, Graph, BallRoot), v_id(R, GraphPrime, BallRootPrime)].

v_id(R, Graph, BallRoot):- @id[ball_size(R, Graph, BallRoot)].

ball_size(R, Graph, BallRoot):- vertex_in_ball(R, Graph, BallRoot, V).
```

c) `@inv[ball_size(Radius, Graph, BallRoot)]` corresponds to the normalization term $1/|V_g|$. `@inv` is the meta-function that computes the multiplicative inverse and `ball_size(Radius, Graph, BallRoot)` is a the float predicate that counts the number of vertices in a `Radius`-neighborhood rooted in `BallRoot`.
d) `k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime,VPrime)` computes k_{INV} using `H_WL` iterations of the WL algorithm to obtain vertex features `phi_wl(R, H_WL, Graph, BallRoot, V)` from the `R`-neighborhood subgraphs.

```
:- declare(k_inv/7, real).
:- declare(phi_wl/5, polynomial(real)).
wl_iterations(3). % constant

k_inv(R, Graph, BallRoot, V, GraphPrime, BallRootPrime, VPrime):-
  wl_iterations(H_WL),
  @dot[phi_wl(R, H_WL, Graph, BallRoot, V),
    phi_wl(R, H_WL, GraphPrime, BallRootPrime, VPrime)].


phi_wl(R, 0, Graph, BallRoot, V):-        phi_wl(R, H, Graph, BallRoot, V):-
  wl_color(R, Graph, BallRoot, 0, V).       H > 0, H1 is H-1,
                                            phi_wl(R, H1, Graph, BallRoot, V).
phi_wl(R, H, Graph, BallRoot, V):-
  H > 0, wl_color(R, Graph, BallRoot, H, V).
```

where `wl_color/5` is defined as `wl_color/3`, but has two additional arguments `R` and `BallRoot` that are needed to restrict the graph connectivity to the `R`-neighborhood subgraph rooted in vertex `BallRoot`.

4.4.3 Random walk graph kernels

Vishwanathan et al (2010) propose generalized random walk kernels. The similarity between a pair of graphs is computed by performing random walks on both graphs and then counting the number of matching walks.

Counting the number of matching random walks between two graphs $G_a = (V_a, E_a)$ and $G_b = (V_b, E_b)$ is equivalent to counting the number of walks in $G_\times = (V_\times, E_\times)$, where $G_\times$ is the direct product between the graphs G_a and G_b [55].

The direct product graph $G_\times$ is defined in terms of G and G' as follows:

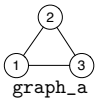
$$\begin{aligned} V_\times &= \{(v_a, v_b) | v_a \in V_a \wedge v_b \in V_b\} \\ E_\times &= \{((v_a, u_a), (v_b, u_b)) | (v_a, u_a) \in E_a \wedge (v_b, u_b) \in E_b\} \end{aligned} \tag{16}$$

To encode the product graph in kProbLog, we start from an edge connectivity predicate `edge_asymm/3` such that `edge(Graph, V, U)` is true whenever there is an edge between vertices `V` and `U` in graph `Graph`. In a similar way we define the vertex predicate `vertex/2`.

```
:- declare(vertex/2, bool).
:- declare(edge_asymm/3, bool).
:- declare(edge/3, real).
```

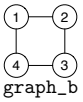
Predicate `edge/3` when its first argument is `graph_a` (`graph_b`) represents the adjacency matrix $W_a \in \mathbb{R}^{|V_a| \times |V_a|}$ ($W_b \in \mathbb{R}^{|V_b| \times |V_b|}$) of graph G_a (G_b).

We shall now consider the same example graphs used by Vishwanathan et al (2010) starting from two graphs G_a and G_b encoded with the kProbLog symbols `graph_a` and `graph_b`.



graph_a

```
vertex(graph_a, 1).     edge_asymm(graph_a, 1, 2).
vertex(graph_a, 2).     edge_asymm(graph_a, 1, 3).
vertex(graph_a, 3).     edge_asymm(graph_a, 2, 3).
```



graph_b

```
vertex(graph_b, 1).     edge_asymm(graph_b, 1, 2).
vertex(graph_b, 2).     edge_asymm(graph_b, 2, 3).
vertex(graph_b, 3).     edge_asymm(graph_b, 3, 4).
vertex(graph_b, 4).     edge_asymm(graph_b, 1, 4).
```

we then define `edge/3` as the symmetric closure of `edge_asymm/3`.

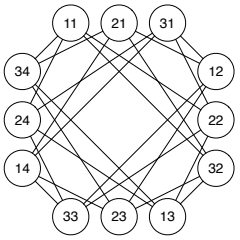
```
:- declare(edge/3, real).
edge(Graph, V, W):- edge_asymm(Graph, V, W).
edge(Graph, V, W):- edge_asymm(Graph, W, V).
```

The kernel definition also includes starting $\mathbf{p}_a \in \mathbb{R}^{|V_a|}$ ($\mathbf{p}_b \in \mathbb{R}^{|V_b|}$) and stopping $\mathbf{q}_a \in \mathbb{R}^{|V_b|}$ ($\mathbf{q}_b \in \mathbb{R}^{|V_b|}$) probabilities associated to the vertices of the graph G_a (G_b), that we shall assume to be uniform.

```
:- declare(prob_start/2, real).
:- declare(prob_stop/2, real).
:- declare(graph_size/1, real).

graph_size(G):- vertex(G, V).

prob_start(G, V):- % uniform initial probability
  vertex(G, V), @inv[graph_size(G)].

prob_stop(G, V):- % uniform stopping probability
  vertex(G, V), @inv[graph_size(G)].
```

The product graph $G_\times$ can be specified following Equation 16. When the first argument of predicate `edge/3` is `kron(graph_a, graph_b)` it represents the adjacency matrix $W_\times = W_a \times W_b \in \mathbb{R}^{|V_a||V_b| \times |V_a||V_b|}$ of $G_\times$.

According to Vishwanathan et al (2010) also the initial (stopping) probabilities $\mathbf{p}_\times$ ($\mathbf{q}_\times$) of the vertices $V_\times$ can be obtained as the Kronecker product between the initial (stopping) probabilities of G_a and G_b, i.e. $\mathbf{p}_\times = \mathbf{p}_a \times \mathbf{p}_b$ ($\mathbf{q}_\times = \mathbf{q}_a \times \mathbf{q}_b$).



kron(graph_a, graph_b)

```
vertex(kron(Ga, Gb), i(Va, Vb)):-
  vertex(Ga, Va), vertex(Gb, Vb).

edge(kron(Ga, Gb), i(Va, Vb), i(Ua, Ub)):-
  edge(Ga, Va, Ua), edge(Gb, Vb, Ub).

prob_start(kron(Ga, Gb), i(Va, Vb)):-
   prob_start(Ga, Va), prob_start(Gb, Vb).

prob_stop(kron(Ga, Gb), i(Va, Vb)):-
   prob_stop(Ga, Va), prob_stop(Gb, Vb).
```

The above definition of Kronecker product differs from the Kronecker product given in Section 3.1, only in the parametrization of the connectivity with graph identifiers.

The generalized random walk kernel [55] is expressed as:

$$k(G, G') = \sum_{k=0}^{\infty} \mu(k) \mathbf{q}_\times^\top W_\times^k \mathbf{p}_\times \tag{17}$$

where $W_\times^k$ is the k^{th} power of $W_\times$. The element related to the `i(Va, Vb)`$^{\text{th}}$-row and `i(Ua, Ub)`$^{\text{th}}$-column of $W_\times^k$ represents the similarity between simultaneous length k random walks [55]. While $\mu(k)$ is a factor that weighs the contribution, that the paths of length k give to the similarity.

Different definitions of the parameter $\mu(k)$ lead to different instances of random walk graph kernels. We specify in kProbLog the geometric variant. The geometric random walk graph kernel between two graphs G and G' is obtained by setting $\mu(k) = \lambda^k$.

$$k(G, G') = \mathbf{q}_\times^\top \sum_{k=0} \lambda^k W_\times^k \mathbf{p}_\times = \mathbf{q}_\times^\top (I - \lambda W_\times)^{-1} \mathbf{p}_\times. \tag{18}$$

Vishwanathan et al (2010) propose different methods to compute such kernel value. For our kProbLog specification we choose fixed-point iterations in which Equation 18 is rewritten as:

$$k(G, G') = \mathbf{q}_\times^\top \mathbf{x} \tag{19}$$

$$(I - \lambda W_\times)\mathbf{x} = \mathbf{p}_\times. \tag{20}$$

where $\mathbf{x}$ is an unknown and can be solved using the iterative update rule [55]:

$$\mathbf{x}_{t+1} = \mathbf{p}_\times + \lambda W_\times \mathbf{x}_t. \tag{21}$$

until the fixed point is reached. We shall assume that $\lambda = 0.5$ and specify the iterative update of Equation 21 as:

```
:- declare(lambda/0, real).
:- declare(x_sol/2, real, destructive).
:- declare(geometric_rw_kernel/2, real).
0.5::lambda.
x_sol(kron(Ga, Gb), i(Va, Vb)):-
  vertex(Ga, Va), vertex(Gb, Vb), randn(0, 0.001).

lambda_w_product_x(GraphKron, I):-
  lambda, edge(GraphKron, I, J), x_sol(GraphKron, J).

x_sol(GraphKron, I):-
  @addition[lambda_w_product_x(GraphKron, I),
  p_product(GraphKron, I)].
```

The geometric random walk graph kernel is then specified according to Equation 21 as:

```
geometric_rw_kernel(Ga, Gb):-
  q_product(kron(Ga, Gb), I),
  x_sol(kron(Ga, Gb), I).
```

5 kProbLog$^\epsilon$: dual numbers for algorithmic differentiation

The need for accurately computed derivatives is ubiquitous and algorithmic differentiation (AD) [30] has emerged as a very useful tool in many areas of Science. In particular, AD has occupied a special role in machine learning [3] since the introduction of the Backpropagation algorithm for training neural networks. Recent advances in deep learning have led to a proliferation of many frameworks for AD

such as Torch [6], Theano [2] and TensorFlow [1]. While it is beyond the scope of this paper to develop an alternative to these frameworks for deep learning, we show in this Section how to use the semiring of dual numbers and the gradient semiring (a generalization of dual numbers) [16, 34] in kProbLog for AD.

A dual number $x + \epsilon x'$ consists of a primal part x and a dual part x' where ϵ is the nilpotent element (i.e. $\epsilon^2 = 0$). In the case of variables $x' = 1$ while in the case of constants $x' = 0$.

Example 6 Let $f(x)$ and $g(x)$ be two real valued functions over reals with derivatives $f'(x)$ and $g'(x)$ respectively. We have the following rules for the derivative of combined functions:

1. *sum rule* $\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$,
2. *product rule* $\frac{d}{dx}(f(x)g(x)) = f(x)g'(x) + f'(x)g(x)$.

We use dual numbers and represent f and g together with their derivatives as $y = f(x) + \epsilon f'(x)$ and $z = g(x) + \epsilon g'(x)$.

According to the algebra of dual numbers we have that:

1. *sum* $y + z = f(x) + g(x) + \epsilon(f'(x) + g'(x))$,
2. *product* $yz = f(x)g(x) + \epsilon(f(x)g'(x) + f'(x)g(x))$.

We observe that the dual part of $y+z$ and yz are the combined-function derivatives that we obtained with the *sum rule* and the *product rule* respectively.

Dual numbers are generalized to gradients by introducing multiple nilpotent elements $\epsilon_1, \ldots, \epsilon_n$ such that $\epsilon_i \epsilon_j = 0,\ \forall i, j$. The gradient number $x + \epsilon_1 x'_1 + \ldots + \epsilon_n x'_n$ combines the primal part x with n partial derivatives $x'_1, \ldots, x'_n$.

In kProbLog we denote the nilpotent element ϵ with the compound term `eps(index_term)`, where the argument `index_term` is some term that is used to index distinct partial derivatives. The meta-function `@grad/2` takes as inputs a dual number y and a nilpotent element ϵ_x and outputs the partial derivative $\frac{\partial y}{\partial x}$.

Example 7 Differentiation of a quadratic form. Let us assume that we have a quadratic form $f(\mathbf{x}) = \mathbf{x}^\top A\mathbf{x}$ where $A = \left[\begin{smallmatrix} 2 & 1 \\ 6 & 3 \end{smallmatrix}\right]$ and we want to compute its gradient ∇f in $\mathbf{x}_0 = \left[\begin{smallmatrix} 2 \\ 1 \end{smallmatrix}\right]$.

```
:- declare([x0/1, a/2], float).
:- declare(grad_f/1, float).
:- declare(dim/1, term).
:- declare([x/1, f/0], grad).

2::a(0, 0). 1::a(0, 1).
6::a(1, 0). 3::a(1, 1).

2::x0(0). 1::x0(1).

x(I):- x0(I).
eps(I)::x(I):- range(I, 0, 2).
```

```
f :-
  a(I, J), x(I), x(J).

eps(Dim)::dim(Dim):-
  range(Dim, 0, 2).

grad_f(I):-
  range(I, 0, 2),
  @grad[f, dim(I)].

query(grad_f(_)).
```

Where we defined `x(I)` as `x0(I)`$+\epsilon$.

The output of this program is:

```
20.0::grad_f(1).
15.0::grad_f(0).
```

The gradient semiring adds to kProbLog support for AD and can naturally be employed for gradient descent learning.

A very natural task to express in kProbLog is matrix factorization [45, 37, 33].

Example 8 Koren et al (2009) propose a basic factorization model. Users and items are mapped to a joint f-dimensional factor space. The interaction between an item and a user is modeled as the inner-product between their representations in the factor space.

Each user u is associated with a vector $\mathbf{q}_u \in \mathbb{R}^f$ while each item i is associated with a vector $\mathbf{p}_i$ and r_{ui} is the rating given by user u to item i. The goal is to approximate the rating r_{ui} with a score derived by the inner-product between $\mathbf{q}_u$ and $\mathbf{p}_i$. Koren et al (2009) use the mean squared error between the predicted score and the rating r_{ui} and regularize the factor representations of users and items ($\mathbf{q}_u$ and $\mathbf{p}_i$) with the ℓ_2-norm.

In kProbLog we can represent $\mathbf{q}_u$, $\mathbf{p}_i$ and r_{ui} declaring the predicates:

```
:- declare([q/2 p/2], grad).
:- declare(r/2, real).
```

The rating predicate `r/2` is initialized according to the available data, while the initial weight of `q/2` and `p/2` will be a dual number whose primal number is initialized with small random values (to break symmetries) and whose dual part ϵ_f is identified by a nilpotent element indexed by the factor index i.e.:

```
p(Item, Factor):- randn(0, 0.001).
eps(Item, Factor)::p(Item, Factor).

q(User, Factor):- randn(0, 0.001).
eps(User, Factor)::p(User, Factor).
```

The cost predicate `cost/0` is defined in terms of `q/2` and `p/2` and then is differentiable.

```
:- define(lambda/0, real).
:- define([score/2, cost_mse/2, cost_reg_user/0, cost_reg_item/0, cost/0], grad).

1::lambda. % COST HYPERPARAMETER
score(User,Item):-
  q(User, Factor), p(Item, Factor).

cost_mse(User, Item):- % MEAN SQUARED ERROR
  @square[@subtract[r(User, Item), score(User, Item)]].

cost_reg_user:- % L2-REGULARISER
  @square[q(User, Factor)].

cost_reg_item:- % L2-REGULARISER
  @square[p(Item, Factor)].

cost:- cost_mse(User, Item).
cost:- lambda, cost_reg_user.
cost:- lambda, cost_reg_item.
```

In the above program we specified the cost `cost/0` function to minimize as a sum of the mean squared error `cost_mse(User, Item)` and an ℓ_2-norm regularizer `cost_reg(User, Item)` weighted by a hyper-parameter `lambda` that we set to 1. by default. We can express `cost/0` using mathematical formulae as follows:

$$cost = \sum_{ui} \underbrace{(r_{ui} - score_{ui})^2}_{\texttt{cost\_mse(User, Item)}} + \lambda \underbrace{\left(\sum_{u,f} q_{uf}^2 \right)}_{\texttt{cost\_reg\_user}} + \underbrace{\left(\sum_{i,f} p_{if}^2 \right)}_{\texttt{cost\_reg\_item}} . \quad (22)$$

The optimization of such a cost function can be performed with gradient descent.

6 kProbLog$^{D[\mathbb{S}]}$: ProbLog and aProbLog as special cases

We now clarify the relationship between kProbLog and Problog, and we show that the ProbLog implementation using SDDs of [56] can be emulated by kProbLog. ProbLog is a probabilistic programming language that defines a probability distribution over possible worlds (Herbrand interpretations). A ProbLog program consists of a set of definite clauses c_i and a set of facts labeled with probabilities p_i. While a Prolog query can either succeed or fail, ProbLog computes the success probability of a query. The success probability of a query is the sum of the probabilities of all the possible worlds I in which the query q is true, it thus corresponds to the probability that it is true in a randomly chosen possible world.

Fig. 3 Example of a ProbLog program (on the left) with the enumeration of the possible worlds and their probabilities (on the right).

```
0.5::p(a).
0.6::p(b).
p(c) :- p(a), p(b).
p(d) :- p(a).
p(d) :- p(b).

query(p(_)).
```

Worlds in which `p(c)` is true.

{p(a), p(b)}	0.6×0.5	$= 0.3$
	p(c)	$= 0.3$

Worlds in which `p(d)` is true.

{p(a)}	$0.5 \times (1 - 0.6)$	$= 0.2$
{p(b)}	$(1 - 0.5) \times 0.6$	$= 0.3$
{p(a), p(b)}	0.6×0.5	$= 0.3$
	p(d)	$= 0.8$

Example 9 In Figure 3 (on the left) we show a ProbLog program in which there are two facts `p(a)` and `p(b)` with probability labels 0.5 and 0.6 respectively. `p(c)` and `p(d)` are defined as the conjunction `p(a)` $\wedge$ `p(b)` and the disjunction of `p(a)` $\vee$ `p(b)` respectively. On the right we show two tables which compute the probabilities of `p(c)` and `p(d)`. For `p(c)` we have one possible world while for `p(d)` there are three possible worlds. For both `p(c)` and `p(d)` we enumerate the worlds in which they are true and compute their weighted model count.

To compute the probabilities of queries, ProbLog compiles the logical part of the program into a Boolean circuit and then evaluates this circuit on the probabilities p_i. The circuit is evaluated by replacing disjunctions and conjunctions with sums and products respectively. The compilation process is necessary to cope with the disjoint-sum problem [12, 34]. For instance, to compute $P($`p(d)`$)$ we cannot simply sum up $P($`p(a)`$)$ and $P($`p(b)`$)$ (two possible explanations/proofs for `p(d)`) as this would lead to a value that is larger than one, but rather we need to compute $P($`p(a)`$) + P($`p(b)` $\wedge \neg$ `p(a)`$)$. The disjoint-sum problem can be solved by representing the Boolean circuit either as a decision diagram. In practice this can be an ordered binary decision diagram (OBDD) [4] or as an SDD [8]. While the first version of ProbLog [12] was using OBDDs a more recent work [56] used SDDs.

The key property that makes OBDDs suitable to handle the disjoint-sums problem is *determinism* [9] which guaranties that conjunctions in OBDDs are

mutually exclusive. SDDs are a strict superset of OBDDs which maintains their key property such as determinism, canonicity and polytime composability [8].

Algebraic model counting (AMC) generalizes probabilistic model counting to a semiring S. In kProbLog it is possible to employ a semiring $D[\mathbb{S}]$ to specify AMC tasks on an arbitrary commutative semiring $\mathbb{S}$. The semiring of valued decision diagrams $D[\mathbb{S}]$ can be represented using an SDD whose variables are labeled with elements from the commutative semiring $\mathbb{S}$. Valued decision diagrams are similar to PSDD [36], except that values are not necessarily probabilities and they do not necessarily encode probability distributions.

Any ProbLog program can be directly translated into a kProbLog$^{D[\mathbb{R}]}$ program using the semiring of SDDs labeled with probabilities. This is a direct consequence of the fact that the evaluation algorithm of kProbLog generalizes the T_P-compilation with SDDs of Vlasselaer et al (2015) to arbitrary semirings. If we label kProbLog facts with SDDs we recover the compilation algorithm of Vlasselaer et al (2015).

Example 10 We now compare the same program specified in ProbLog with the probability semiring (i.e. ProbLog) and in kProbLog$^{D[\mathbb{R}]}$ with SDDs labeled with probabilities:

ProbLog

```
0.5::p(a).
0.6::p(b).
p(c) :- p(a), p(b).
p(d) :- p(a).
p(d) :- p(b).

query(p(_)).
```

kProbLog$^{D[\mathbb{R}]}$

```
:- declare(p, sdd(real)).
sdd(0.5, p(a))::p(a).
sdd(0.6, p(b))::p(b).
p(c) :- p(a), p(b).
p(d) :- p(a).
p(d) :- p(b).

query(p(_)).
```

The semiring values `sdd(0.5, p(a))` and `sdd(0.6, p(b))` represent parametrized SDD variables and are in one-to-one correspondence with kProbLog facts.

The notation `sdd(Value,Atom)::Atom` used for kProbLog is cumbersome and can be replaced by the syntactic sugar `Value::Atom`. In this way the kProbLog$^{D[\mathbb{R}]}$ program becomes syntactically identical to the ProbLog one.

So far we have shown that kProbLog$^{\mathbb{R}}$ can perform probabilistic model counting. This behavior is not enforced by the language as in ProbLog, but is optional (i.e. it is induced by the type declaration `:- declare(p, sdd(real))`).

While kProbLog$^{D[\mathbb{R}]}$ is equivalent to ProbLog, it is also straightforward to represent aProbLog on a semiring S as kProbLog$^{D[\mathbb{S}]}$ using SDDs labeled with semiring values.[7]

Algebraic model counting is useful for inference tasks and reasoning about possible worlds, but there are some tasks which are nontrivial to express in aProbLog. Examples are linear algebra operations and explicit feature extraction as explained in Section 4.1.

[7] Probabilities have *neutral sums* (i.e. for each atom a we have that $p(a) + p(\neg a) = 1$) but this property is not verified for semirings in general. This issue is known as the *neutral-*

7 Experimental evaluation

We now experimentally evaluate kProbLog and show how it can be used as a declarative language for machine learning. The choices that a kProbLog programmer needs to make in order to satisfy a requirement are quite different from the ones that an imperative programmer would do. While an imperative programmer would have to use different data structures to meet the software requirements, a kProbLog user can just specify the requirements with logical rules. For instance, when moving from a directed to an undirected graph, imperative programmers would have to change their data structure, while in kProbLog it suffices to simply add an extra rule to capture the symmetry of undirected graphs.

kProbLog is well suited for prototyping. As we will show below with an example of graph kernel (cf. **E2**), kProbLog makes it easy to compose existing programs in order to construct new ones. Different graph kernels take into account different structural aspects. For example, the WL subtree kernel can capture degree centrality while shortest path kernels do not. On the other hand, shortest path kernels are a natural choice if one wants to capture patterns with distant nodes. Even though the WL subtree patterns can capture distant nodes, the number of iterations required to do so could lead to diagonal dominant kernels. Since both these kernels can easily be specified in kProbLog (as we will show), it is also straightforward to create a hybrid graph kernel combining the strengths of both underlying kernels.

Another powerful construct of kProbLog are the meta-functions. In a machine learning context, meta-functions can be exploited as a flexible and expressive instrument for describing rich families of kernels. In this sense, meta-functions can be interpreted as a powerful generalization of common kernel hyperparameters, lifting them from simple numbers to functions. We will show in **E1** how meta-functions can be exploited to explore multiple feature spaces against the same logical specification and provide a rich class of feature spaces.

Our experiments address the following questions:
Q1 Can we use meta-functions to explore multiple feature spaces against the same kProbLog specification and increase the classification accuracy?
Q2 Can kProbLog produce hybrid kernels that combine the strengths of existing ones?
Q3 Are the results obtained with kProbLog in line with the state of the art?

7.1 Datasets

We empirically validate some kProbLog specifications on the following natural language and chemical datasets:

QC [40] is a dataset about question classification and contains 5500 training and 500 test questions from the TREC10 QA competition. Question classifiers are often used to improve the performance of question answering systems. Indeed, they can be used to provide constraints on the answer types and determine answer selection strategies. QC labels questions according to a two-layer taxonomy of answer types. The taxonomy contains 6 coarse classes (ABBREVIATION,

sums problem [34]. Kimmig et al (2011) explain how to overcome the *neutral-sums problem* by modifying the evaluation of a Boolean circuit.

ENTITY, DESCRIPTION, HUMAN, LOCATION and NUMERIC VALUE) and 50 fine classes.

Example 11 The sentence "What films featured the character Popeye Doyle ?" is labeled in QC as ENTITY since we expect films in the answer.

In order to be comparable with the existing literature, we adopted the coarse grained labels as classification targets.

MUTAG [14] is a dataset of 188 mutagenic compounds labeled according to whether or not they have a mutagenic effect on the Gramnegative bacterium Salmonella typhimurium.

BURSI [32] is dataset of 4337 molecular compounds subdivided in two classes (2401 mutagens and 1936 nonmutagens) determined with the Ames in vitro assay.

7.2 Experiments

E1 This experiment was designed to provide an answer to **Q1** and, in particular, to illustrate the expressiveness of kProbLog's meta-functions in an NLP context, where a large number of options are typically available to describe the feature space. It also aims to answer **Q3** since question classification is a typical task where good results using graph kernels have been reported in the literature [40, 58]. Each sentence in the QC dataset is represented as a sequence of tokens. For this purpose, we define a predicate `token_labels/1`. `token_labels/1` is a unary relation that associates to each token `t` an algebraic label which encodes word, lemma and part of speech (POS) tag of token `t`.

We then use a dependency parser [10][8] to extract typed dependency relations between tokens and encode them using the predicate `dep_rel/2`. `dep_rel/2` encodes an edge in the graph of the dependency relations of a sentence, the type of the dependency relation is encoded as algebraic label. The dependency relations between the tokens of the sentence in Example 11 are encoded as:

```
x(det)::dep_rel(1, 0).
x(nsubj)::dep_rel(2, 1).
x(dobj)::dep_rel(2, 6).
x(punct)::dep_rel(2, 7).
x(det)::dep_rel(6, 3).
x(compound)::dep_rel(6, 4).
x(compound)::dep_rel(6, 5).
```

We then define a predicate `dep_rel_edge/2` that casts dependency edges `dep_rel(V, W)` to the shortest path semiring and defines shortest paths on the dependency graph with the predicate `spath/2` (see the definition of the shortest path semiring in Appendix Section B).

```
:- declare([spath/2, dep_rel_edge/2], shortest_paths , additive).
dep_rel_edge(V, W):- @cast_to_shortest_path[dep_rel(V, W)].
spath(V, W):- dep_rel_edge(V, W).
spath(V, W):- V != W, dep_rel_edge(V, U), spath(U, W).
```

We used the `@cast_to_shortest_path/1` meta-function to cast `dep_rel/2` to the shortest path semiring predicate `dep_rel_edge/2`.

We extract unigram and shortest path features with the rules:

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[token_labels(V), config].
feature_blocks:- @decorate_paths[spath(V, W), v2labels, config].
```

[8] We used the spaCy Python library to extract lemmas, POS tags and dependency relations.

the meta-functions `@decorate_vertices` and `@decorate_paths` replace the token indices in unigrams and paths with token labels (i.e. words, lemmas, POS tags or no label) according to the information specified by the algebraic label of the `config` atom. The `config` atom also specifies whether or not edge labels should be placed in the decorated shortest paths. Since the algebraic label of the `config` atom encodes a set of configurations, the output of `@decorate_vertices` and `@decorate_paths` is a multiset of feature blocks represented by the semiring `polynomial(polynomial(real))` that associates a block of features to each possible configuration specified by the algebraic label of `config`.

Finally all the feature blocks are aggregated together into the algebraic label of `final_features` which corresponds to the feature vector of the sentence.

```
:- declare([final_features/0], polynomial(real)).
final_features:- @aggregate_feature_blocks[feature_blocks].
```

We used the above kProbLog specification to extract features for 127 different configurations (i.e. value assignments of the algebraic label of atom `config`). We considered two kinds of structural features: unigrams and shortest paths. Tokens can be labeled with words (w), lemmas (l), POS tags (p) or not labeled at all (_). There are 8 ways of generating features by combining blocks of unigrams labeled with words, lemmas and POS tags. The possible unigram configurations correspond to the power set of $\{w, p, l\}$. Similarly we have 16 possible ways of combining blocks of shortest path features in which the token indices are replaced by an appropriate token label of type $\{w, p, l, \_\}$, for the p and _ types we also include the edge label information (we add the edge label between labels of consecutive tokens). From the Cartesian product of unigram and shortest path configurations we obtain a total of 128 from which we skip the one without feature blocks. We ran classification experiments using a linear SVM classifier with the C parameter set to 10^4. In Table 1 we report the classification accuracy of the top 16 best configurations.

Table 1 List of the 16 configurations that achieve the highest classification accuracy on QC.

unigram features	shortest path features	test accuracy	unigram features	shortest path features	test accuracy
lw	*lp*	91.6%	*lpw*	*_pw*	90.4%
lpw	*lp*	91.2%	*lpw*	*_lp*	90.4%
lw	*_lpw*	91.0%	*pw*	*_l*	90.2%
lw	*_lw*	90.6%	*lw*	*_pw*	90.2%
lw	*_lp*	90.6%	*lpw*	*pw*	90.2%
lpw	*_lpw*	90.6%	*lw*	*pw*	90.0%
lw	*lpw*	90.4%	*lpw*	*_lw*	90.0%
lpw	*lpw*	90.4%	*w*	*pw*	89.8%

We measured the runtime of the feature extraction and we found that none of the 127 runs on QC exceeds 32 seconds. The measurement of the runtime was performed on a 16 cores machine (Intel Xeon CPU E5-2665@2.40GHz and 96GB of RAM).

E2 In this experiment we mainly aim to answer **Q2** and, in particular, to test the ability of kProbLog to hybridize two well known graph kernels in a context (molecule classification) where they are known to perform well. In order to capture

the complementary advantages of the WL subtree and shortest path kernels, mentioned at the beginning of this section, we shall specify a hybrid kernel. We extract histograms of shortest paths and decorate them with WL labels. This is where we hybridize the two kernels. The reader should not confuse this kernel with the WL shortest path kernel [53] (explained in Section 4.4.1) which takes as features pairs of WL labels together with their shortest path distance.

We encode each molecule in the MUTAG dataset with the `vertex/1` and the `edge_asymm/2` predicates which represent atoms labeled with atom symbols and chemical bonds labeled with their type respectively. Differently the graph of the dependency relations of a sentence, molecules are naturally represented as undirected graphs so we define the predicate `edge/2` as the symmetric closure of `edge_asymm/2`.

```
:- declare([edge/2], polynomial(real)).
edge(V, W):- edge_asymm(V, W).
edge(V, W):- edge_asymm(W, V).
```

We use the `@cast_to_shortest_path/1` meta-function to cast `edge/2` to the shortest path semiring and generate shortest paths.

```
:- declare([spath/2, edge_sp/2], shortest_paths , additive).
edge_sp(V, W):- @cast_to_shortest_path[edge(V, W)].
spath(V, W):- edge_sp(V, W).
spath(V, W):- V != W, dep_rel_edge(V, U), spath(U, W).
```

We generate the WL labels of the vertices in the graph.

```
:- declare([wl/2, wl_multiset/2], polynomial(real)).
wl(0, V):- @id[vertex(V)].
wl_multiset(H, V):- edge(V, W), wl(H, W).
wl(H, V):- 0 < H, H <= MAX_ITER, @id[wl(H-1, V), wl_multiset(H-1, V))].
```

We create a predicate `v2wl/1` whose atoms `v2wl(H)` associate to the $\mathtt{H}^{\text{th}}$ iteration of the WL algorithm a dictionary that maps vertices `V` of the graph to their WL feature at step `H`.

```
:- declare([v2wl/1], polynomial(polynomial(real))).
v2wl(H):- wl(H, V), @poly_var[V].
```

where the meta-function `@poly_var/1` creates a polynomial variable $x_{\mathtt{V}}$ indexed by the term `V`.

We decorate vertices and shortest paths in a molecule with WL labels.

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[wl(H, V)].
feature_blocks:- @decorate_paths[spath(V, W), v2wl(H)].
```

And this is the step in which the hybridization happens.

Finally we aggregate the resulting feature blocks using a normalize sum normalize schema.

```
:- declare([normalized_features/0], polynomial(real)).
normalized_features:- @block_normalize_sum_normalize(feature_blocks)
```

For BURSI we use the same kProbLog specification of MUTAG, but we impose `K_SP_MAX`$= 2$ as maximum path length. So, we updated the shortest path predicate `spath/2` to `spath/3` as follows:

```
:- declare([spath/3, edge_sp/2], shortest_paths).
edge_sp(V, W):-
    @cast_to_shortest_path[edge(V, W)].
spath(1, V, W):-
```

```
    edge_sp(V, W).
spath(K, V, W):-
    V != W, 2 <= K, K <= K_SP_MAX, edge_sp(V, U), spath(K-1, U, W).
```

Consequently the `feature_blocks/0` predicate is updated to:

```
:- declare([feature_blocks/0], polynomial(polynomial(real))).
feature_blocks:- @decorate_vertices[wl(H, V)].
feature_blocks:- @decorate_paths[spath(K, V, W), v2wl(H)].
```

For both MUTAG and BURSI we set maximum number of WL iterations to `MAX_ITER`= 1 and ran our kProbLog specification. We made classification experiments using 10 fold cross-validation and measured the classification accuracy and area under the ROC curve for MUTAG and BURSI respectively. We repeated 10 times the 10 fold cross-validations and we obtained an average accuracy of 91.1% with a standard deviation of 0.9% for MUTAG and an average area under the roc curve of 0.902 with a standard deviation of 0.001 for BURSI. For both datasets we used a linear SVM classifier with the C parameter set 1. We measured the runtime on the same hardware used in **E1**. The runtime for MUTAG was 32 seconds while BURSI was 5 minutes and 7 seconds.

All the experiments can be reproduced by running the code provided with the kProbLog implementation (see Section 3.3).

7.3 Discussion

We now answer the experimental questions:

A1 In **E1** we explored a parametrized feature space for QC, using different combinations of words, lemmas, POS tags we could list the 16 best parameterizations in Table 1. Since the best results are in line with the results reported by [40] and [58], we conclude that meta-functions are a valid language construct to parametrize the feature space. The 91.6% of accuracy obtained on QC with the experiments in **E1** is in line with the results reported by [40] and [58].

A2 Shervashidze et al [53] experimented on MUTAG with 8 different graph kernels and achieved the highest accuracy (87.3 ± 0.6) with shortest path kernels, while the accuracy obtained with the WL subtree kernel is 82.1 ± 0.4 (see Table 1 [53]). As anticipated in the beginning of this section, the WL subtree kernel and the shortest path kernel capture different topological aspects. In experiment **E2**, thanks to the declarative nature of kProbLog, we made a hybrid and labeled shortest paths with WL colors. We experimented with this kernel on MUTAG and obtained an accuracy of $91.1 \pm 0.9\%$, which is significantly higher than the ones individually achieved by the shortest path and WL subtree kernels. In **E2**, we also experimented on BURSI with the same hybrid kernel and obtained 0.902 ± 0.001 of area under the ROC, this result is line with those reported in Table 1 of [7].

A3 The 91.6% of accuracy obtained in **E1** on QC are in line with the ones reported by [40] and [58]. The $91.1 \pm 0.9\%$ of accuracy obtained with our hybrid kernel in **E2** on MUTAG is significantly higher than the ones obtained with 8 different graph kernels in [53]. Also the 0.902 ± 0.001 area under the ROC curve obtained in **E1** on BURSI is in line with the results reported by Costa and De Grave (2010). For

these reasons, we conclude that kProbLog can be used to specify kernels that work well on real-world application domains. The runtimes measured are reasonable and show that kProbLog is usable in practice. Feature extraction on QC and MUTAG took less than a minute while on BURSI took less than 6 minutes.

In principle we could reimplement any kProbLog program in a declarative language such as Prolog or an imperative language such as Python, but we would loose flexibility and elegance in both cases. Prolog easily expresses relational data, but in order to handle mathematical labels the user would be forced to code inside the rules not only the relational aspect, but also the algebraic aspect. In this sense, kProbLog is advantageous because it decouples the relational aspect from the algebraic aspect and avoids to write boilerplate code. Imperative languages such as Python, C++ and Java offer rich libraries for scientific computing and machine learning, but do not have builtin support for logical variables and unification. In particular each meta-function in a kProbLog program can be put into one-to-one correspondence with functions (e.g. a Python function). However, these imperative languages do not have the equivalent of meta-clauses which are first-order constructs and support logical variables.

8 Related work

In the introduction, we claimed that kProbLog can express models for tensor-based operations, for kernels, and for probabilistic programs; we also mentioned approaches such as Dyna and aProbLog. We now discuss related work along these lines.

First, kProbLog is able to combine logic with tensors and can express tasks such as matrix factorization. As such kProbLog is related to Tensor Relational Algebra [33], which combines tensors with relational algebra and which was successfully employed for tensor decomposition. However, tensor relational algebra does not support recursion and is therefore less expressive than kProbLog.

Secondly, and perhaps most importantly, kProbLog can be used to declaratively specify a wide range of relational and graph kernels and feature extraction problems using polynomial semirings. As such it is related to the kLog system [22], which has focused on the specification of relational learning problems and provides a framework to map them into graph-based learning problems via a procedure called *graphicalization*. In conjunction with a graph kernel, kLog can construct feature vectors associated with tuples of objects in relational domains. However, kLog does not provide support for *programming* the kernel itself, it uses a built-in kernel (the NSPDK [7]) or defers the kernel specification to external plugins. kLog and kProbLog are therefore complementary languages. Furthermore, by adopting kProbLog in kLog one would obtain a statistical relational learning system in which the kernel could be declaratively specified as well. Also Gärtner et al contributed kernels within a typed higher-order logic in which individuals (the examples) are represented as terms and the kernel definitions, specified in a lambda calculus, exploit the syntactic structure of these example representations. While this also yields a declarative language for specifying kernels on structured objects, it does neither involve the use of semirings nor was it applied to other modeling tasks such as those involving probabilistic reasoning.

Finally, kProbLog is an algebraic logic programming system building upon aProbLog [34] and Dyna [19, 18]. The relationships to these languages are quite subtle and more technical. Nevertheless, distinguishing features of kProbLog are that it supports A) multiple semirings, B) meta-functions, C) additive and destructive updates, D) algebraic model counting, and E) its semantics are rooted in logic programming theory (using an adaptation of the T_P-operator [56]).

On the other hand, aProbLog [34] is a generalization of the probabilistic programming language ProbLog [12] to semirings. ProbLog and other statistical relational learning formalisms are based on a possible world semantics on weighted model counting. The key contribution of aProbLog is that it generalizes weighted model counting to algebraic model counting [35] based on commutative semirings instead of the probabilistic semiring. kProbLog extends aProblog in that it supports multiple semirings (A), meta-functions (B) and destructive as well as additive updates (C). Furthermore, kProbLog (in particular the kProbLog$^{D[\mathbb{S}]}$) replicates aProbLog by performing AMC on a semiring S using the semiring of SDDs whose variables are labeled with values which belong to the semiring S. Furthermore, aProbLog was conceived for algebraic reasoning about possible worlds, while kProbLog main design goal was the specification of tensor algebra and feature extraction problems.

A second closely related language is Dyna [19, 18], a language that was initially conceived as a semiring weighted extension of Datalog for dynamic programming. Dyna has been developed for quite a while and is a fairly complex language supporting many different extensions of the basic algebraic Datalog. While kProbLog builds upon Dyna's ideas, Dyna does not support meta-functions (B), destructive updates (C), and algebraic model counting (D). Concerning (D), Dyna has not dealt with the disjoint-sum problem occurring in probabilistic and algebraic logics such as ProbLog and aProbLog. Furthermore, the semantics of Dyna have been specified in a more informal way in [17] using the definition of a *valuation function* and although [17, 18] relate Dyna's semantics to a T_P-operator; Dyna's T_P-operator is not formally defined in these papers (E).

9 Conclusions

We proposed kProbLog, a simple algebraic extension of Prolog that can be used for declarative machine learning, most importantly, for kernel programming. Indeed, using polynomials and meta-functions allows to elegantly specify many recent kernels (e.g. the WL graph kernel, propagation kernels and GIKs) in kProbLog.

We further introduced in the language the semiring of dual numbers so that kProbLog can also express gradient descent learning, while the semiring of dual numbers allowed us to specify matrix factorization. We showed how the semiring of decision diagrams allows to capture aProbLog (and so ProbLog and, hence, probabilistic programming) as a fragment of kProbLog.

All these features make kProbLog a language in which the user can combine rich logical and relational representations with algebraic ones to declaratively specify models for machine learning. Our experimental evaluations showed that kProbLog can be applied to real world datasets, obtaining good statistical performance and runtimes.

Acknowledgements We would like to thank Angelika Kimmig and Anton Dries for the fruitful discussions about ProbLog.

References

1. Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, Corrado GS, Davis A, Dean J, Devin M, Ghemawat S, Goodfellow I, Harp A, Irving G, Isard M, Jia Y, Jozefowicz R, Kaiser L, Kudlur M, Levenberg J, Mané D, Monga R, Moore S, Murray D, Olah C, Schuster M, Shlens J, Steiner B, Sutskever I, Talwar K, Tucker P, Vanhoucke V, Vasudevan V, Viégas F, Vinyals O, Warden P, Wattenberg M, Wicke M, Yu Y, Zheng X (2015) TensorFlow: Large-scale machine learning on heterogeneous systems. URL `http://tensorflow.org/`, software available from tensorflow.org
2. Bastien F, Lamblin P, Pascanu R, Bergstra J, Goodfellow IJ, Bergeron A, Bouchard N, Bengio Y (2012) Theano: new features and speed improvements. Deep Learning and Unsupervised Feature Learning NIPS 2012 Workshop
3. Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (2015) Automatic differentiation in machine learning: a survey. arXiv preprint arXiv:150205767
4. Bryant RE (1992) Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys (CSUR) 24(3):293–318
5. Ceri S, Gottlob G, Tanca L (1989) What you always wanted to know about datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1(1):146–166, DOI 10.1109/69.43410, URL `http://dx.doi.org/10.1109/69.43410`
6. Collobert R, Bengio S, Mariéthoz J (2002) Torch: a modular machine learning software library. Tech. rep., IDIAP
7. Costa F, De Grave K (2010) Fast neighborhood subgraph pairwise distance kernel. In: Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel, pp 255–262, URL `http://www.icml2010.org/papers/347.pdf`
8. Darwiche A (2011) SDD: A new canonical representation of propositional knowledge bases. In: IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011, pp 819–826, DOI 10.5591/978-1-57735-516-8/IJCAI11-143, URL `http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-143`
9. Darwiche A, Marquis P (2002) A knowledge compilation map. Journal of Artificial Intelligence Research 17(1):229–264
10. De Marneffe MC, Manning CD (2008) The stanford typed dependencies representation. In: Coling 2008: proceedings of the workshop on cross-framework and cross-domain parser evaluation, Association for Computational Linguistics, pp 1–8
11. De Raedt L (2008) Logical and relational learning. Springer Science & Business Media
12. De Raedt L, Kimmig A, Toivonen H (2007) Problog: A probabilistic prolog and its application in link discovery. In: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, pp 2462–2467, URL `http://ijcai.org/Proceedings/07/Papers/396.pdf`

13. De Raedt L, Kersting K, Natarajan S, Poole D (2016) Statistical relational artificial intelligence: Logic, probability, and computation. Synthesis Lectures on Artificial Intelligence and Machine Learning 10(2):1–189
14. Debnath AK, Lopez de Compadre RL, Debnath G, Shusterman AJ, Hansch C (1991) Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. Journal of medicinal chemistry 34(2):786–797
15. Droste M, Kuich W (2009) Semirings and formal power series. Springer
16. Eisner J (2002) Parameter estimation for probabilistic finite-state transducers. In: Proceedings of the 40th annual meeting on Association for Computational Linguistics, Association for Computational Linguistics, pp 1–8
17. Eisner J, Blatz J (2007) Program transformations for optimization of parsing algorithms and other weighted logic programs. In: Proc. of Formal Grammar, pp 45–85
18. Eisner J, Filardo NW (2011) Dyna: Extending datalog for modern ai. In: Datalog Reloaded, Springer, pp 181–220
19. Eisner J, Goldlust E, Smith NA (2004) Dyna: A declarative language for implementing dynamic programs. In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL), Companion Volume, Barcelona, pp 218–221
20. van Emden MH, Kowalski RA (1976) The semantics of predicate logic as a programming language. J ACM 23(4):733–742, DOI 10.1145/321978.321991, URL http://doi.acm.org/10.1145/321978.321991
21. Esparza J, Luttenberger M, Schlund M (2014) Fpsolve: A generic solver for fixpoint equations over semirings. In: Implementation and Application of Automata - 19th International Conference, CIAA 2014, Giessen, Germany, July 30 - August 2, 2014. Proceedings, pp 1–15, DOI 10.1007/978-3-319-08846-4_1, URL http://dx.doi.org/10.1007/978-3-319-08846-4_1
22. Frasconi P, Costa F, De Raedt L, De Grave K (2014) klog: A language for logical and relational learning with kernels. Artif Intell 217:117–143, DOI 10.1016/j.artint.2014.08.003, URL http://dx.doi.org/10.1016/j.artint.2014.08.003
23. Garcez Ad, Besold TR, de Raedt L, Földiak P, Hitzler P, Icard T, Kühnberger KU, Lamb LC, Miikkulainen R, Silver DL (2015) Neural-symbolic learning and reasoning: contributions and challenges. In: Proceedings of the AAAI Spring Symposium on Knowledge Representation and Reasoning: Integrating Symbolic and Neural Approaches, Stanford
24. Garcez AS, Lamb LC, Gabbay DM (2008) Neural-symbolic cognitive reasoning. Springer Science & Business Media
25. Gärtner T, Flach P, Wrobel S (2003) On graph kernels: Hardness results and efficient alternatives. In: Learning Theory and Kernel Machines, Springer, pp 129–143
26. Gärtner T, Lloyd JW, Flach PA (2004) Kernels and distances for structured data. Machine Learning 57(3):205–232
27. Getoor L, Taskar B (eds) (2007) Introduction to statistical relational learning. Adaptive computation and machine learning, MIT Press, Cambridge, Mass
28. Golub GH, Van Loan CF (2012) Matrix computations, vol 3. JHU Press
29. Green TJ, Karvounarakis G, Tannen V (2007) Provenance semirings. In: Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART symposium on Prin-

ciples of database systems, ACM
30. Griewank A, Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition, 2nd edn. Society for Industrial and Applied Mathematics
31. Kashima H, Tsuda K, Inokuchi A (2003) Marginalized kernels between labeled graphs. In: ICML, vol 3, pp 321–328
32. Kazius J, McGuire R, Bursi R (2005) Derivation and validation of toxicophores for mutagenicity prediction. Journal of Medicinal Chemistry
33. Kim M, Candan KS (2011) Approximate tensor decomposition within a tensor-relational algebraic framework. In: Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011, pp 1737–1742, DOI 10.1145/2063576.2063827, URL http://doi.acm.org/10.1145/2063576.2063827
34. Kimmig A, Van den Broeck G, De Raedt L (2011) An algebraic prolog for reasoning about possible worlds. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2011, San Francisco, California, USA, August 7-11, 2011, URL http://www.aaai.org/ocs/index.php/AAAI/AAAI11/paper/view/3685
35. Kimmig A, Van den Broeck G, De Raedt L (2012) Algebraic model counting. CoRR abs/1211.4475, URL http://arxiv.org/abs/1211.4475
36. Kisa D, Van den Broeck G, Choi A, Darwiche A (2014) Probabilistic sentential decision diagrams. In: Proceedings of the 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)
37. Koren Y, Bell R, Volinsky C (2009) Matrix factorization techniques for recommender systems. Computer pp 30–37
38. Kuich W (1997) Semirings and formal power series: Their relevance to formal languages and automata. In: Handbook of formal languages, Springer, pp 609–677
39. Landwehr N, Passerini A, De Raedt L, Frasconi P (2006) kfoil: Learning simple relational kernels pp 389–394
40. Li X, Roth D (2002) Learning question classifiers. In: Proc. of the 19th international conference on Computational linguistics-Volume 1, Association for Computational Linguistics
41. Mahé P, Ueda N, Akutsu T, Perret JL, Vert JP (2004) Extensions of marginalized graph kernels. In: Proceedings of the twenty-first international conference on Machine learning, ACM, p 70
42. Milch B, Marthi B, Russell SJ, Sontag D, Ong DL, Kolobov A (2005) BLOG: probabilistic models with unknown objects pp 1352–1359, URL http://ijcai.org/Proceedings/05/Papers/1546.pdf
43. Muggleton S, Raedt LD, Poole D, Bratko I, Flach PA, Inoue K, Srinivasan A (2012) ILP turns 20 - biography and future challenges. Machine Learning 86(1):3–23, DOI 10.1007/s10994-011-5259-2, URL http://dx.doi.org/10.1007/s10994-011-5259-2
44. Neumann M, Patricia N, Garnett R, Kersting K (2012) Efficient graph kernels by randomization. In: Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part I, pp 378–393, DOI 10.1007/978-3-642-33460-3_30, URL http://dx.doi.org/10.1007/978-3-642-33460-3_30

45. Nickel M, Tresp V, Kriegel HP (2011) A three-way model for collective learning on multi-relational data. In: Proceedings of the 28th international conference on machine learning (ICML-11), pp 809–816
46. Nilsson U, Maluszynski J (1990) Logic, programming and Prolog. Wiley
47. Orsini F, Frasconi P, De Raedt L (2015) Graph invariant kernels. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pp 3756–3762, URL http://ijcai.org/Abstract/15/528
48. Quinlan JR (1990) Learning logical definitions from relations. Machine learning 5(3):239–266
49. Richardson M, Domingos PM (2006) Markov logic networks. Machine Learning 62(1-2):107–136, DOI 10.1007/s10994-006-5833-1, URL http://dx.doi.org/10.1007/s10994-006-5833-1
50. Sammut C (1993) The origins of inductive logic programming: A prehistoric tale. In: Proceedings of the 3rd International Workshop on Inductive Logic Programming, J. Stefan Institute, pp 127–147
51. Sato T (1995) A statistical learning method for logic programs with distribution semantics. In: Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995, pp 715–729
52. Sato T, Kameya Y (1997) PRISM: A language for symbolic-statistical modeling. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes, pp 1330–1339, URL http://ijcai.org/Proceedings/97-2/Papers/078.pdf
53. Shervashidze N, Schweitzer P, van Leeuwen EJ, Mehlhorn K, Borgwardt KM (2011) Weisfeiler-lehman graph kernels. Journal of Machine Learning Research 12:2539–2561, URL http://dl.acm.org/citation.cfm?id=2078187
54. Van Laer W, De Raedt L (2001) How to upgrade propositional learners to first order logic: A case study. In: Machine Learning and Its Applications, Springer, pp 102–126
55. Vishwanathan SVN, Schraudolph NN, Kondor R, Borgwardt KM (2010) Graph kernels. The Journal of Machine Learning Research 11:1201–1242
56. Vlasselaer J, Van den Broeck G, Kimmig A, Meert W, De Raedt L (2015) Anytime inference in probabilistic logic programs with tp-compilation. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pp 1852–1858, URL http://ijcai.org/Abstract/15/263
57. Whaley J, Avots D, Carbin M, Lam MS (2005) Using datalog with binary decision diagrams for program analysis. In: Programming Languages and Systems, Springer
58. Zhang D, Lee WS (2003) Question classification using support vector machines. In: SIGIR 2003: Proceedings of the 26th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, July 28 - August 1, 2003, Toronto, Canada, pp 26–32, DOI 10.1145/860435.860443, URL http://doi.acm.org/10.1145/860435.860443

A Appendix: proof of Theorems 1 and 2

Proof (of Theorem 1) In line 4 of Algorithm 1 the ground program GROUND(P) is subdivided into n strata, where n is finite and never exceeds the total number of ground atoms in GROUND(P). Strata are visited in sequence (lines 5-26), for each stratum the for loop (lines 14-26) applies the algebraic T_P-operator exactly once for each ground acyclic rule, then the loop at lines 14-26 produces no side effects Algorithm 1 and terminates. The loop on cyclic rules (lines 14-26) does not produce side effects on w, because the loops at lines 16, 18 and 21 are never executed since *CYCLIC* is empty for acyclic programs.

Proof (of Theorem 2) The proof of Theorem 2 is identical to the one of Theorem 1 except that *CYCLIC* is not empty for some strata. We just need to prove that when a stratum P_i is a cyclic kProbLog$^{\mathbb{S}_i}$ program on an ω-continuous semirings $\mathbb{S}_i$ the loop at lines 14-26 terminates. Since when there are no meta-functions, lines 15-25 implement an update of the atom weights w according to Eq 2 which corresponds to a step of the Kleene iteration in a system of polynomial equations. Because $\mathbb{S}_i$ is ω-continuous the termination of the loop at lines 14-26 is guaranteed by Proposition 1.

B Shortest path semiring

The shortest path semiring is a variant of the tropical semiring that keeps track of the set of shortest paths corresponding to a given shortest path distance.

The elements $a \in \mathcal{S}$ of the shortest path semiring $(\mathcal{S}, \oplus, \otimes, 0_s, 1_s)$, are sets of strings over the vocabulary V of the vertex identifiers. All the strings in a must have the same length $len(a)$

Let $a, b \in \mathcal{S}$ sum and product are defined as follows:

$$a \oplus b = \begin{cases} a & \text{if } len(a) < len(b) \\ b & \text{if } len(a) > len(b) \\ a \cup b & \text{if } len(a) = len(b) \end{cases} \tag{23}$$

$$a \otimes b = \{\text{CONCAT}(s_a, s_b) | s_a \in a \wedge s_b \in b\} \tag{24}$$

where CONCAT is the string concatenation operator.

Additive and multiplicative identity are the empty set $\emptyset$ and the singleton set $\{\epsilon\}$ containing the empty string ϵ respectively.